Fortran 95 y paralelismo con HPF2

Jorge D'Elía
<jdelia@intec.unl.edu.ar>

www: http://www.cimec.org.ar/calculoparalelo
Facultad de Ingeniería y Ciencias Hídricas
Universidad Nacional del Litoral http://fich.unl.edu.ar
Centro Internacional de Métodos Computacionales en Ingeniería
http://www.cimec.org.ar

(Document version: hpf-2.0)
(Date: 2006/09/29 06:20:00 UTC)

Indice

Sobre este apunte:

Este apunte corresponde a uno de los temas del curso *Cálculo Científico con Computadoras Paralelas* dictado por Victorio E. Sonzogni, Mario A. Storti y Jorge D'Elía, como curso de posgrado del programa de *Doctorado en Ingeniería* de la *Facultad de Ingeniería y Ciencias Hídricas* (http://www.fich.unl.edu.ar) de la *Universidad Nacional del Litoral* (http://www.unl.edu.ar).

Página web del curso: La página web del curso es http://www.cimec.org.ar/calculoparalelo.

Utilitarios usados: Todo este apunte ha sido escrito con utilitarios de *software* libre, de acuerdo a los lineamientos de la *Free Software Foundation/GNUProject* (http://www.gnu.org). La mayoría de los utilitarios corresponden a un sistema `GNU/Red Hat Linux release 8.0 (Psyche) Kernel 2.4.18-14 on an i686`.

- El apunte ha sido escrito en LaTeX y convertido a PDF con `pdflatex`. El mismo está completamente inter-referenciado usando las utilidades propias de LaTeX y el paquete `hyperref`.
- Los ejemplos en F95 han sidos desarrollados y probados con el compilador `GNU g95` (Linux x86 Binary (2005-06-04 08:58)) (http://g95.sourceforge.net/)
- Las figuras han sido generadas con *Xfig 3.2.3d* (http://www.xfig.org/).

Agradecimientos A Mario A. Storti por facilitar los fuentes *.tex que sirvieron como punto de partida.

Errores Si encuentra algún error en el apunte le agradecemos reportarlo al autor, indicando la versión del mismo, tal como aparece en la portada.

Capítulo 1

Preliminares

En estas notas se resumen algunas técnicas básicas para la programación en un *cluster Beowulf* mediante el lenguaje Fortran F95 mediant la extensión *High Performance Fortran* (HPF).

Nota: otras alternativas con Fortran a nuestro alcance en un cluster Beowulf serían, por ejemplo, el Message Passing Interface (MPI) o bien el Parallel Virtual Machine (PVM) para el paso de mensajes entre nodos. En el caso de un cluster Beowulf construido con Pentium duales también sería posible usar además la norma Open Machine Parallel [16] (OMP) para hacer paralelismo dentro de cada Pentium dual pero, necesariamente, con paso de mensajes entre nodos.

Como lenguaje Fortran asumiremos el especificado por las normas Fortran 90 y Fortran 95, abreviados aquí como F90 y F95, respectivamente, hasta donde lo admita el compilador empleado. Omitiremos el Fortran 2003 (F03) pues todavía no hay disponibles compiladores de distribución libre, mientras que a nivel comercial mencionamos por ejemplo el `NAGware 95` [12], versión 5.1. Tampoco consideraremos al Fortran 2008 (F08) y su extensión *Co-Array* que ha sido propuesta como otro paradigma de paralelismo pues es de muy reciente discusión. En lo que sigue se asume un conocimiento rudimentario del F90-F95 tal como el disponible en las referencias dadas en la Sec. 1.3. De todas maneras, primero se hace un breve repaso de F90-F95 y luego se considerará el HPF. En los demos se ha tratado de comentar todo lo posible acerca de lo que se hace. Por ejemplo, en los detalles quiźas más "oscuros", por ejemplo, el uso de las operaciones de dispersión de índices dados en la Sec. 2.6.

Notar que uno puede darse una idea de cómo andar en bicicleta leyendo un texto de ciclismo pero será mucho más instructivo si se dispone de una bicicleta y espacio donde practicar. Del mismo modo, se puede tener una idea de cómo programar en paralelo consultando algún texto del tema, pero es mucho más instructivo si el lector dispone además de un *cluster Beowulf* y de un compilador con los cuales programar. Se incluyen *demos* pues en programación también se aprende mirando lo ya hecho e inclusive puede ser útil reusar partes ya hechas. Sin embargo, los mismos no constituyen un estado del arte ni mucho menos pues en algunos casos se han sacrificado eficiencia y brevedad para intentar una presentación más clara.

Con respecto de la "utilidad" de aprender un poco de HPF diremos que (i) para un programador nativo de Fortran es probablemente lo más sencillo como para empezar en un *cluster Beowulf*, y (ii) una buena parte del paradigma asumido por el mismo se puede implementar mediante otras extensiones normalizadas de los lenguajes de programación orientados a cómputo científico. Por ejemplo, la extensión *Open Machine Parallel*[16] (OMP) para C++/F95. De hecho, la Norma OMP v2.5 (Mayo

2005) unifica la exposición e incluye los mismos demos tanto para C++ como para F95. Por otra parte, el *US-ISO Fortran Committee* decidió en Mayo de 2005 incluir *co-arrays* en la siguiente revisión 2008 del Fortran. El mismo es una extensión del F95, con sintaxis similar a la de arreglos en Fortran, con notación explícita más simple para la descomposición de los datos (comparado con los modelos basado en pasos de mensajes), independiente de la arquitectura y, en principio, apto tanto en máquinas de memoria distribuida como compartida, incluyendo *clusters* [13].

1.1. Introducción

Fortran es un lenguaje estándar (norma ISO) orientado al álgebra matricial numérica, dentro de un contexto más general de arreglos con más de dos índices. La versión F03 es completamente orientada a objetos mientras que sus precursoras F90 y F95 lo son sólo parcialmente, e.g. no contemplan ni herencia *dinámica* ni funciones *virtuales*. Muy convencionalmente diremos que las normas F90, F95 y F03 surgieron en los años 1990, 1995 y 2003, respectivamente, si bien las fechas efectivas de cada norma suelen ser muy posteriores. Las diferencias entre el F90 y el F95 son más bien pocas pero no despreciables. Por ejemplo, entre otras, mencionamos: `forall`, `where`, `cpu_time` y la inicialización `null()` en la declaración de punteros. Algunas de las instrucciones secuenciales (o seriales) en F90-F95 preveen la eventualidad de hacerlas en forma distribuida (o en paralelo) de un modo automático cuando se compila en un multiprocesador (e.g. la instrucción `forall`). Empero, el F90-F95 no proveen sintaxis para que el programador pueda controlar la distribución ni de los datos ni de las tareas.

En cambio, el HPF es *una* extensión estandarizada del Fortran que permite acceder a diferentes grados de paralelismo a medida que fue evolucionando. Por ejemplo, las normas HPF1.x estuvieron exclusivamente orientadas al modelo del paralelismo en los *datos*, i.e. los datos son distribuidos sobre un conjunto de procesadores y se efectúa la misma operación sobre cada parte, lo cual también es conocido como un modelo *Single Program Multiple Data* (SPMD). Más recientemente, desde las normas HPF2.x, se está incluyendo gradualmente el paralelismo en las *tareas*, es decir, diferentes tareas en diferentes procesadores.

En estas notas se considera solamente un repertorio bastante restringido de la sintaxis HPF2 posible. Hay varios items que no se discuten por razones de espacio y para no duplicar lo disponible en el *manual de usuario* y en el *manual de referencia* del compilador HPF empleado, e.g. los provistos en el traductor `ADAPTOR`[5] que emplearemos durante esta parte del curso.

1.2. Algunos compiladores F90-F95, C++ y HPF

Entre los compiladores que más se han experimentado últimamente tanto en los *cluster Beowulf* como en las PC individuales del CIMEC, bajo Linux, citamos los siguientes:

1. Proyecto *GNU*: compiladores `g++`, `g77` y `g95` para C++, F77 y F95, respectivamente. Los dos primeros son clásicos que ya suelen venir preinstalados en muchos paquetes de distribución del *Linux* [25], mientras que el restante es de más reciente disponibilidad en internet [14]. Notar que el `gfortran` es *gcc-native*, es decir, es parte del proyecto general *GNU Compiler Collection* (GCC), a diferencia del `g95` [14] que es *gcc-based*, esto es, es un desarrollo independiente no-integrado oficialmente. Los compiladores `g77` y `g95` son exclusivamente *secuenciales* (o seriales),

i.e. no preveen paralelismo de ningún tipo, mientras que muy recientemente el `g++` ha empezado a incluir la norma OMP;

2. *Intel*[15]: compiladores `icc` e `ifort` para C++/OMP y F95/OMP, respectivamente, de libre distribución bajo la modalidad *General Public license* (GPL) en *Linux*, pero únicamente con licencia comercial bajo *windows*. Está bastante evolucionado en su soporte de la norma F95 y más "amistoso" para un usuario final. Entre otras características, incluye la opción de precisión extendida de tipo **cuádruple**. Su histórica compatibilidad con la norma OMP lo hace conveniente para aquellos programadores interesados en emplear, por ejemplo, una *Pentium dual* o en un *cluster* de *Pentium duales*. En este último caso notemos que sólo podremos hacer paralelismo con OMP dentro de cada nodo pero, obligadamente, con paso de mensajes entre los nodos; Por otra parte, esta empresa ofrece su *Intel Cluster Toolkit* con una instalación muy "aceitada" con todo lo necesario para un *cluster Beowulf*, aunque sin un compilador HPF.

3. `ADAPTOR`[5]: es un *traductor* para la norma HPF2 bajo *Linux*, de libre distribución y orientado específicamente a los *cluster Beowulf* de diversas arquitecturas. Traduce y compila programas fuentes de HPF2 o bien de F77-F95, incluyendo la eventualidad de OMP. Este traductor se apoya en compiladores secuenciales y en alguna librería de paso de mensajes, e.g. con el *ifort* y el `MPICH2` que es como lo hemos armado en los *clusters* del CIMEC. Su importancia radica en que, junto con el traductor de origen japonés `FHPF`[22] son los únicos proyectos activos de libre distribución que implementan la norma HPF2, mientras que los demás, prácticamente, han sido discontinuados;

4. `Portland PGI`[33] vende diversos paquetes de compiladores de C++, F95 y HPF, incluyendo un *kit* específicamente orientado a los *cluster Beowulf*. Su compilador HPF recientemente ha implementado la norma HPF2, y es un compilador nato y no un traductor, como en el caso del ADAPTOR, lo cual siempre es mucho mejor.

1.3. Un desglose de la bibliografía

- Lenguaje de programación F90-F95 (básico): Marshall-Morgan-Schonfelder[27], Marshall-Schonfelder[28], Corde-Delouis[7], Ewing-Hare-Richardson[11];
- Paralelismo con HPF: Marshall[26], Koebel[24] *et al.*, Ewing[11] *et al.*, Brandes[4, 5];
- Fundamentos teóricos de cálculo en paralelo: Cormen-Leiserson-Rivest[8], Parhami[31], Foster[17], Grama-Grupta-Karypis-Kumar[20].
- Algoritmos numéricos en general: Press[34] *et al.*, Arndt[1], Chu-George[6], De la Fuente O'Connor[9];
- Lenguaje de programación F90-F95 (con aplicaciones varias): Meissner[29], Norton[30], Decyk[10];
- Cálculo numérico y paralelismo en C++ y MPI: Karniadakis-Kirby[23];
- Paralelismo con OMP: norma OMP para los lenguajes F95 y C++[16], Graham[19], Hermans[21].

1.4. Convenciones en la notación

- Los programas fuente tendrán extensión ya sea `*.f`, `*.f90` o `*.hpf` (hay que mirar el manual del compilador empleado);
- En los fragmentos de código usaremos casi siempre minúsculas (ya sea palabras `reservadas` del lenguaje, identificadores, funciones y procedimientos del usuario). Tener cuidado porque F77-F95 *no distinguen* entre mayúsculas y minúsculas;
- En los programas fuente emplearemos casi exclusivamente el formato `libre` (equivalente al empleado en los lenguajes Pascal o C). A veces, podría aparecer el "antiguo" formato `fijo`. Notar que el formato *libre* es el recomendado a partir del F90;
- En los programas fuente con formato libre se coloca un signo de exclamación (!) para indicar que desde ahí hasta el final de la línea es un comentario. La línea tiene en principio 72 caracteres pero es frecuente que los compiladores soporten alguna longitud extendida. Por ejemplo, el `ifort` admite 132;
- Muchos de los acrónimos utilizados aquí se mantendrán en idioma inglés y se recopilan en la Tabla 1.1 junto con el significado de otras abreviaturas más frecuentes.

1.5. Sobre la versión 2006

Estas notas se modifican permanentemente en base a la experiencia que se va ganando en el dictado del tema y por las actualizaciones en el lenguaje F95-HPF o en los compiladores empleados, de modo que a través de los años hay algunos temas que cambian, otros se eliminan y otros que se agregan.
En esta ocasión se actualizaron algunos items apuntando a la norma HPF2 y al traductor `ADAPTOR`, por ejemplo, en la Sec. 3.17 se resume cómo compilar con `ADAPTOR` y luego cómo lanzar la corrida en un *cluster Beowulf* usando algunos de los comandos más básicos de `MPICH2`.
Por último, agradezco a Laura Battaglia por hacerme notar las diversas erratas y sugerencias para mejorar la exposición de algunas secciones.

Acrónimo	Significado
F77	Fortran 77
F90	Fortran 90
F95	Fortran 95
F03	Fortran 2003
F08	Fortran 2008
GLP	General Public license
HPF	High Perfomance Fortran
HPF1	HPF según normas 1.x
HPF2	HPF según normas 2.x
MMP	Massively Multiprocesador machine
MPI	Message Passing Interface
MPMD	Multiple Program Multiplie Data
OMP	Open Machine Parallel
POO	Programación Orientada a Objetos
PVM	Parallel Virtual Machine
SPMD	Single Program Multiple Data
SSP	Small-Scale Parallel machine
lg(n)	logaritmo en base 2 de n
R^n	espacio euclídeo de n dimensiones
Abreviatura	**Significado**
i.e.	es decir (esto es), del latín *id est*
e.g.	por ejemplo, del latín *exempli gratia*
LHS	*Left Hand Side* (miembro izquierdo)
RHS	*Right Hand Side* (miembro derecho)
1D	1 dimensión
2D	2 dimensiones

Tabla 1.1: Acrónimos y abreviaturas frecuentes.

Capítulo 2

Ejemplos de sintaxis matricial en F90-F95

El Fortran es un lenguaje de programación diseñado para computación científica. En su forma estándar, omitiendo las extensiones OMP y HPF, sólo se prevee un cierto grado de *paralelismo en los datos*. Las normas F90 y F95 se basan en el "viejo" Fortran 77 (F77) en donde, entre otros cambios, ha sido mejorada la sintaxis para la manipulación de los arreglos multi-índice (de hasta 7). En estas notas, a veces, referenciaremos a los arreglos multi-índice como arreglos "matriciales" y también diremos simplemente F90-F95 englobando ambas normas, haciendo sólo la distinción cuando haga falta. El F90-F95 dispone de una extensa librería intrínseca matricial y Tipos Abstractos de Datos (TAD), aunque nada equivalente a la STL del C++. Asimismo, son sólo parcialmente orientadas a objetos, e.g. no soportan herencia dinámica ni funciones virtuales de un modo natural. El F90 es un salto cualitativo con respecto del F77, entre otras cosas, brinda la notación de arreglos similar a `Octave` o al `Scilab`. Como ya mencionamos en la introducción, las diferencias entre el F90 y el F95 son más bien pocas pero no despreciables. Por ejemplo, entre otras, destacamos que sólo desde F95 se cuenta con: `forall`, `where`, `cpu_time` y la inicialización `null()` en la declaración de punteros. Por otra parte, el Fortran 2003 es un salto cualitativo pues se exige que admita una completa orientación a objetos. Empero, todavía no existe un compilador de libre distribución (GPL) que lo implemente en los cambios más significativos, mientras que entre los compiladores de tipo comercial, sólo conocemos el `NAGware 95` [12], versión 5.1.

2.1. Algunas erratas sorprendentes en F90-F95

Lo que sigue ha sido adaptado a partir de la colección de erratas sorprendentes en el uso del F90 que han sido recopiladas por Szymanski [37]. Los compiladores, en diverso grado, suelen fallar en el diagnóstico de los siguientes *bugs*:

- problemas con variables declaradas e inicializadas simultáneamente;
- sorpresas con las funciones y procedimientos genéricos;
- problemas con el `intent(out)` en tipos derivados;
- problemas con los argumentos opcionales;

- erróneo uso del estilo F90;
- peligro con interfases desde F90 a rutinas F77.

2.1.1. Problemas con variables declaradas e inicializadas simultáneamente

Hagamos la suma de 1 a 5 invocando a una función `suma` que es codificada en tres formas distintas:

```
module m_sumar
  public
  contains
    !declara e inicia variable "s" <--> implica un atributo "save"
    integer function suma1 (n)  result (z)
      integer, intent (in) :: n
      integer :: i, s = 0
      do  i = 1, n  ; s = s + i ; end do  ;  z = s
    end function
    !declara variable "s" con un atributo "save"
    integer function suma2 (n)  result (z)
      integer, intent (in) :: n
      integer :: i
      integer, save :: s = 0
      do  i = 1, n  ; s = s + i ; end do  ;  z = s
    end function
    !unica re-inicializacion correcta de la variable "s"
    integer function suma3 (n)  result (z)
      integer, intent (in) :: n
      integer :: i, s
      s = 0
      do  i = 1, n  ; s = s + i ; end do  ;  z = s
    end function
end module
program foo5
  use m_sumar
  print *, "erronea declaracion e inicializacion de una variable "
  print *, "suma1 (1:5) = ", suma1 (5)
  print *, "suma1 (1:5) = ", suma1 (5)
  print *, "suma2 (1:5) = ", suma2 (5)
  print *, "suma2 (1:5) = ", suma2 (5)
  print *, "suma3 (1:5) = ", suma3 (5)
  print *, "suma3 (1:5) = ", suma3 (5)
end program
```

Explicación: una variable local que es inicializada cuando es declarada tiene implícitamente el atributo `save`. Esto significa que la misma es inicializada sólo en la primera llamada de la función, mientras que en las sub-siguientes retiene su último estado. Esto es una verdadera sorpresa para los programadores de C. Para evitar confusiones, si lo que se quiere es recordar el último estado, entonces sería mejor agregar en forma redundante el atributo `save` cuando se declara y se inicializa simultáneamente. Pero si que quiere es re-inicializar cada vez que se llama a la función entonces hay que hacerlo explícitamente cada vez que se entra a la función.

2.1.2. Problemas con intent(out) en tipos derivados

Supongamos una estructura de usuario `utipo` en la cual se definen la variable entera `x` y la variable real `y`. Y que, luego de inicializarlas, pretendemos actualizar sólo una de ellas invocando un procedimiento. Por ejemplo, queremos modificar sólo la variable `x` usando la subrutina `actualiza` de la siguiente manera

```
program test
  type utipo
    integer :: x
    real    :: y
  end type
  type (utipo) :: a
  a.x = 1  ;  a.y = 2.0
  call actualiza (a)
  !ATENCION: a.y PODRIA ESTAR INDEFINIDO AQUI
  print *, a
  contains
    subroutine actualiza (this)
    type (utipo), intent (out) :: this
    !CODIFICACION ERRONEA
    this.x = 2
    end subroutine
  subroutine actualiza (this)
    type (utipo), intent (out) :: this
    !ESTE ES UNA FORMA POSIBLE
    this.x = 2  ;  this.y = 2.0
  end subroutine
  subroutine actualiza (this)
    type (utipo), intent (inout) :: this
    !ESTE ES OTRA FORMA POSIBLE
    this.x = 2
  end subroutine
end program
```

Nota: por problemas en la edición hemos cambiado el operador % por el punto. Explicación: el problema aquí es que cuando se usa `intent(out)` para un tipo derivado, las demás componentes que no fueron asignadas en ese procedimiento podrían quedar indefinidas al salir. Por ejemplo, aún cuando `a.y` estaba definida al entrar a la subrutina `actualiza`, podría quedar indefinida al salir pues nunca fue asignada ahí. La moraleja es que si usamos el `intent(out)`, entonces *todas* las componentes de un tipo derivado deberían ser asignadas. Lo mejor es imaginar que el `intent(out)` se comporta como la variable resultado de una función, es decir, todas las componentes del valor de retorno de la función debe ser asignadas. La alternativa más cómoda y segura es usar `intent(inout)`.

2.1.3. Problemas con los argumentos opcionales

Cuando se usan argumentos opcionales, por brevedad, se suele codificar en un estilo como el indicado como codificación "riesgosa" en el siguiente demo

```
subroutine imprime1 (this, eco)
  character (len=*), intent (in) :: this
  logical, optional, intent (in) :: eco
  !CODIFICACION RIESGOSA
  if (present (eco) .and. eco) then
    print 10, this
  endif
  10 format (a)
end subroutine
subroutine imprime2 (this,eco)
  character (len=*), intent (in) :: this
  logical, optional, intent (in) :: eco
  !CODIFICACION PRECAVIDA
  if (present(eco)) then
     if (eco) print 10, this
  endif
  10 format (a)
end subroutine
program test
  call imprime2 ('esto es un eco', .true.)
end
```

Explicación: la primera forma en `imprime1` no es segura pues quizás el compilador empleado no use la convención del "cortocircuito", esto es, en una pregunta con un `and`, si la primera condición fuera falsa entonces no evalúa la segunda. En este caso, el compilador podría evaluar primero el argumento `eco` antes que la intrínseca `present`. Por eso, es más precavido usar la segunda forma mostrada en `imprime2`.

2.1.4. Sorpresas con las funciones y procedimientos genéricos

Las funciones genéricas en F90 permiten que un mismo nombre de función o subrutina, se use con diferentes tipos de argumentos. Por ejemplo, consideremos el siguiente fragmento

```
    subroutine  first_sub (a,i)
      real    :: a
      integer :: i
    end subroutine
    subroutine second_sub (i,a)
      integer :: i
      real    :: a
    end subroutine
```

Como el primer argumento es real en la primera subrutina y entero en la segunda se diría que ambas son distingibles y, por tanto, podríamos definir un procedimiento genérico haciendo

```
interface first_or_second
  module procedure first, second
end interface
```

Pero esto es peligroso pues en F90 se permite que los procedimientos sean también llamados por los nombres mudos `keywords`) de los argumentos haciendo, por ejemplo,

```
real    :: b
integer :: n
call first_or_second (i=n,a=b)
```

lo cual es problemático pues los procedimientos `first_sub` y `second_sub` son *indistinguibles* en sus argumentos mudos. Es decir, también podríamos haberlas llamado con

```
call first_sub  (i=n,a=b)
call second_sub (i=n,a=b)
```

por lo que la función genérica no quedará bien definida. Una función genérica debe ser capaz de distinguir sus argumentos mudos por tipo y nombre. La solución es usar diferentes argumentos mudos en cada procedimiento, por ejemplo,

```
    subroutine  first_sub (a1,i1)
      real    :: a1
      integer :: i1
    end subroutine
    subroutine second_sub (i2,a2)
      integer :: i2
      real    :: a2
    end subroutine
```

2.1.5. Erróneo uso del estilo F90

En la transición de F77 a F90 a veces se escribe algo como lo que se muestra a continuación

```
program test        !ERRONEO
  integer, dimension (5) :: x
  x = 0 ; call incb (x)
  print *, x
end program
subroutine incb(a)
  integer, dimension (:) :: a
  a = a + 1
end subroutine
```

Aquí, la subrutina `incb` usa un estilo F90 para el arreglo de forma asumida `x`, cuando se declara el `dimension (:)`. Entonces, la subrutina *debe* estar o bien en *módulo* o bien tener una *interfase*, es decir,

```
!PLAUSIBLE
program test2
  interface
    subroutine incb (a)
      integer, dimension (:) :: a
    end subroutine incb
  end interface
  integer, dimension (5) :: x
  x = 0
  call incb(x)
  print *, x
end program main
subroutine incb (a)
  integer, dimension (:) :: a
  a = a + 1
end subroutine incb
```

Para tal fin es mucho más cómodo usar un `module`, pues las interfases son generadas automáticamente por el compilador. Por ejemplo, simplemente tendríamos

```
!MEJOR
module inc
  contains
  subroutine incb (a)
    integer, dimension (:) :: a
    a = a + 1
  end subroutine incb
end module
program test3
  use inc
  integer, dimension (5) :: x
  x = 0
  call incb (x)
  print *, x
end program
```

2.1.6. Peligro con interfases desde F90 a rutinas F77

A veces, en un programa F90 se quiere incluir una subrutina o función que ya fue escrita en F77, por ejemplo,

```
program test
  integer, dimension (5) :: x
  !interfase a una rutina F77
  interface
     subroutine suma (a,n)
     integer :: n
     !ERRONEO para F77
     integer, dimension (:) :: a
     !ESTANDAR para F77
     integer, dimension (n) :: a
     end subroutine
  end interface
  x = 0
  call suma x,5)
  print *, x
end program
c rutina estilo F77
      subroutine suma (a,n)
      dimension a (n)
      do  10 j = 1, n
      a (j) = a (j) + 1
   10 continue
      return
      end
```

La interfase verifica el sincronismo de la llamada y de la declaración de la subrutina. En este caso, la interfase se hace desde un programa F90 a una rutina F77 que tiene un arreglo de forma explícita. Si en la interfase no se da la misma información que la definida en la subrutina F77, el compilador no puede verificar si los argumentos coinciden en tipo, rango y tamaño. En general, las interfases a rutinas F77 deben usar sólo el estilo F77.

2.2. Diferencias entre las instrucciones `do` y `forall`

Un lazo `forall` permite hacer asignaciones más generales que un lazo `do`, por ejemplo,

```
program test
  implicit none
  integer, parameter :: m=2, n=3
  real, dimension (m,n) :: x, y, z
  integer :: i, j
  forall (i=1:m, j=1:n)        x (i,j) = i + j
  forall (i=1:n, j=1:n, i<m)  y (i,j) = 0.0
  forall (i=1:n)               z (i,i) = 1.0
end program
```

en donde se asignan a cada elemento de la matriz `x`, la suma de sus índices, cero a la parte triangular superior de la matriz `y`, y un 1 en la diagonal principal de la matriz `z`, respectivamente. En cada uno de estos tres ejemplos podríamos haber usado alternativamente un lazo `do`. Pero no siempre un lazo `forall` es equivalente a un lazo `do`. Como primer contra-ejemplo, consideremos el vector `a=[11 22 33 44 55]` y las siguientes tareas

```
do      i=2,5  ; a (i) = a (i-1)  ; end do
forall  (i=2:5)  a (i) = a (i-1)
```

i	a
0	[11 22 33 44 55]
2	[11 11 33 44 55]
3	[11 11 11 44 55]
4	[11 11 11 11 55]
5	[11 11 11 11 11]

Tabla 2.1: Ejemplo de la ejecución secuencial del `do`.

El *orden* de cómputo del lazo `do` es *secuencial* y lo resumimos en la Tabla 2.1. En cambio, en el lazo `forall` (ver también la Sec. 3.13.1) se procede con las siguientes etapas:

1. se determina el conjunto de índices *válido* i_v definido por el lazo, en este caso $i_v = (2\ 3\ 4\ 5)$;
2. se determina el conjunto de índices *activo* i_a debido a eventuales máscaras booleanas. En el ejemplo, como el conjunto de índices activo coincide con el conjunto válido, o sea $i_a = (2\ 3\ 4\ 5)$;
3. se evalúa *simultáneamente todas* las expresiones del miembro derecho para los índices activos en *cualquier* orden (pero *sin* hacer todavía las diversas asignaciones). En el ejemplo: `a (i-1)` con `i=2:5`, o sea $(a_1\ a_2\ a_3\ a_4)$=(11 22 33 44);

4. se asigna *toda* la expresión evaluada al miembro izquierdo (en cualquier orden): `a (i)` con `i=2:5`, o sea $(a_2\ a_3\ a_4\ a_5)$=(11 22 33 44).

Vemos que en este caso los resultados son diferentes. Notar que

- dentro del `forall` sólo puede hacer un asignamiento a cada elemento y no puede haber instrucciones condicionales como `where`;
- a *cada* elemento de LHS sólo se le puede asignar un *único* elemento del RHS;
- deben usarse todas las variables indiciales que definen el `forall`, por lo que no puede haber notación con ausencia de índices, es decir, no puede aparecer el operador `:` (dos puntos).

Otro ejemplo que destaca nítidamente la diferencia semántica entre el `do` y el `forall` es obtener la matriz traspuesta del arreglo $\mathtt{A} \in R^{3\times 3}$. Con `forall` basta la operación de trasponer los índices (línea 6) pero, si optamos usar un lazo `do`, entonces necesitaremos un doble lazo y un auxiliar para el *swap* de cada valor (líneas 8-12)

```
1  program traspuestas
2    implicit none
3    integer, dimension (3,3) :: a = reshape ([1,2,3,4,5,6,7,8,9],[3,3])
4    integer :: i, j, t
5    do i = 1, 3 ; print *, "a", (a (i,j),j=1,3) ; end do
6    forall (i=1:3,j=1:3) a (i,j) = a (j,i)
7    do i = 1, 3 ; print *, "b", (a (i,j),j=1,3) ; end do
8    do  i = 1, 3
9      do j = (i+1), 3
10       t = a (i,j) ; a (i,j) = a (j,i) ; a (j,i) = t
11     end do
12   en d do
13   do i = 1, 3 ; print *, "c", (a (i,j),j=1,3) ; end do
14 end program
```

2.3. Sobre el orden de los lazos anidados DO más rápido

En caso de lazos `do` anidados sobre elementos de un arreglo de varios índices, hay que tener en cuenta que en Fortran se almacena por columnas, a diferencia del lenguaje C en donde primero se almacena por filas. Por eso, el orden de los lazos más conveniente en Fortran, en cuanto a una mayor velocidad de cálculo, es el de anidarlos de forma tal que se ejecute primero el lazo sobre el primer índice y luego los lazos sobre los subsecuentes índices. Muchas veces los compiladores, cuando se activan las banderas de optimización, incluyen transformaciones de este tipo en forma automática, pero no siempre es así. Por ejemplo, en el *manual de usuario* del `ADAPTOR` se advierte que el traductor respeta el orden predefinido de los lazos anidados. En el siguiente demo se muestra el orden más conveniente de los lazos anidados en un arreglo de tres índices

```
program test
  implicit none
  integer, parameter :: p=128, q=256, r=512
  integer, dimension (p,q,r) :: a=1, b=2, c
  integer :: i, j, k
  do  k = 1, r
    do  j = 1, q
      do  i = 1, p
        c (i,j,k) = a (i,j,k) + b (i,j,k)
      end do
    end do
  end do
end program
```

2.4. Sintaxis matricial para el producto matriz banda-vector

Por ejemplo, consideremos el siguiente producto matriz-vector,

$$b = Ax = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 \\ a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + a_{35}x_5 \\ a_{43}x_3 + a_{44}x_4 + a_{45}x_5 \\ a_{54}x_4 + a_{55}x_5 \end{bmatrix} . \tag{2.1}$$

Guardemos la matriz banda `A` con el formato comprimido

$$C = \begin{bmatrix} 0 & a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} & 0 \\ a_{54} & a_{55} & 0 & 0 \end{bmatrix} ; \tag{2.2}$$

de n filas y $m = m_1 + 1 + m_2$ columnas, donde m_1 es el número de sub-diagonales, m_2 es el número de supra-diagonales, y el 1 tiene en cuenta la columna de la diagonal principal. En este ejemplo tendremos: $m_1 = 1$ sub-diagonal, $m_2 = 2$ supra-diagonales y $m = 4$ diagonales en total. Empezamos, haciendo la siguiente instrucción matricial:

$$X = \text{spread (x, dim=2, ncopies = m)} = \begin{bmatrix} x_1 & x_1 & x_1 & x_1 \\ x_2 & x_2 & x_2 & x_2 \\ x_3 & x_3 & x_3 & x_3 \\ x_4 & x_4 & x_4 & x_4 \\ x_5 & x_5 & x_5 & x_5 \end{bmatrix} ; \tag{2.3}$$

Para obtener un desplazamiento en las columnas de X, nos conviene introducir el vector auxiliar

$$u = \text{prog_arit (-m1, 1, m)} = [\ -1 \quad 0 \quad 1 \quad 2\ ] ; \tag{2.4}$$

donde `prog_arit` es una función vectorial del usuario que nos devuelve los m primeros elementos de la progresión artimética que empieza en $-m1$ y que va con incremento 1, dada por

```
function prog_arit (v_inicial, tamanio, delta)
  integer, intent (in) :: v_inicial, tamanio, delta
  integer, dimension (tamanio) :: prog_arit
  integer :: k
  prog_arit = 0
  if (tamanio > 0) prog_arit (1) = v_inicial
  do  k = 2, tamanio
    prog_arit (k) = prog_arit (k-1) + delta
  end do
end function
```

De ese modo, a partir de X y de u obtendremos

$$Y = \text{eoshift (X, dim=1, shift = u)} = \begin{bmatrix} 0 & x_1 & x_2 & x_3 \\ x_1 & x_2 & x_3 & x_4 \\ x_2 & x_3 & x_4 & x_5 \\ x_3 & x_4 & x_5 & 0 \\ x_4 & x_5 & 0 & 0 \end{bmatrix} ; \tag{2.5}$$

por lo que el producto *punto* (elemento a elemento) es

$$Z = CY = \begin{bmatrix} 0 & a_{11}x_1 & a_{12}x_2 & a_{13}x_3 \\ a_{21}x_1 & a_{22}x_2 & a_{23}x_3 & a_{24}x_4 \\ a_{32}x_2 & a_{33}x_3 & a_{34}x_4 & a_{35}x_5 \\ a_{43}x_3 & a_{44}x_4 & a_{45}x_5 & 0 \\ a_{54}x_4 & a_{55}x_5 & 0 & 0 \end{bmatrix} ; \tag{2.6}$$

y si lo sumamos por columnas

$$b = \text{sum (Z, dim=2)} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 \\ a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + a_{35}x_5 \\ a_{43}x_3 + a_{44}x_4 + a_{45}x_5 \\ a_{54}x_4 + a_{55}x_5 \end{bmatrix} ; \tag{2.7}$$

nos dará el producto buscado. Todo lo anterior puede resumirse en las instrucciones:

```
n = size (c, dim=1) ; m = size (c, dim=2) ; m1 = ralo % m1
b = sum (c * eoshift ( spread (x, dim=2, ncopies=m), dim=1,
                       shift = prog_arit (-m1,1,m)), dim=2)
```

Comentario: tal estilo de codificación "apretada" suele encontrarse en las librerías de cómputo científico, e.g. en el F90 del *Numerical Recipes* [34].

2.5. Clasificación matricial por incrementos decrecientes

Por ejemplo, supongamos que queremos ordenar de menor a mayor el vector

$$a = [\ 14 \quad 7 \quad 12 \quad 5 \quad 10 \quad 3 \quad 8 \quad 6 \quad 13 \quad 4 \quad 11 \quad 2 \quad 9 \quad 1\ ]$$

de longitud $n = |a| = 14$, utilizando el método de los incrementos decrecientes o *Shellsort* (Shell es el nombre del autor del algoritmo) en una versión matricial *á la* Numerical Recipes[34] dada por el siguiente fragmento

```
subroutine  qshell (a)
  implicit none
  integer, dimension   (:), intent (inout) :: a
  integer, dimension (:,:), allocatable    :: t
  integer, dimension (2)                   :: molde
  integer, dimension (size(a)) :: g
  integer :: i, k, n, p
  n = size (a)
  g = huge (a) !arreglo inicializado con el maximo entero representable
  i = n / 2 ; i = 2 * i
  do while (i > 1)
    i = i / 2
    p = (n + i - 1) / i
    molde = [ i, p ]
    allocate ( t (i,p) )
    t = reshape (source = a, shape = molde, pad = g)
    do while ( any (t (:,1:p-1  ) > t (:,2:p  ) ) )
       call swap (  t (:,1:p-1:2) , t (:,2:p:2)     &
                 , t (:,1:p-1:2) > t (:,2:p:2) )
       call swap (  t (:,2:p-1:2) , t (:,3:p:2)     &
                 , t (:,2:p-1:2) > t (:,3:p:2) )
    end do
    a = reshape (t, shape (a) )
    deallocate (t)
  end do
end subroutine
```

donde el procedimiento `swap` intercambia o no pares de valores de `a,b` según la máscara booleana `mask`, haciendo

```
subroutine swap (a,b,mask)
  implicit none
  integer, dimension (:,:), intent (inout) :: a, b
  logical, dimension (:,:), intent (ino    :: mask
  integer, dimension (size(a,1),size(a,2)) :: t
  where (mask) ; t = a ; a = b ; b = t ; end where
end subroutine
```

Una prueba de escritorio nos conduce a las siguientes etapas:

$$a = \begin{bmatrix} 14 \\ 7 \\ 12 \\ 5 \\ 10 \\ 3 \\ 8 \\ 6 \\ 13 \\ 4 \\ 11 \\ 2 \\ 9 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 14 & 6 \\ 7 & 13 \\ 12 & 4 \\ 5 & 11 \\ 10 & 2 \\ 3 & 9 \\ 8 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 6 & 14 \\ 7 & 13 \\ 4 & 12 \\ 5 & 11 \\ 2 & 10 \\ 3 & 9 \\ 1 & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 6 \\ 7 \\ 4 \\ 5 \\ 2 \\ 3 \\ 1 \\ 14 \\ 13 \\ 12 \\ 11 \\ 10 \\ 9 \\ 8 \end{bmatrix} ; \qquad (2.8)$$

en donde clasificamos por filas en forma reiterada. Pero, como todavía no quedó totalmente clasificada, volvemos a hacer otra pasada pero ahora con otro formato para el arreglo auxiliar `t`

$$t = \begin{bmatrix} 6 & 5 & 1 & 12 & 9 \\ 7 & 2 & 14 & 11 & 8 \\ 4 & 3 & 13 & 10 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} 5 & 6 & 1 & 12 & 9 \\ 2 & 7 & 11 & 14 & 8 \\ 3 & 4 & 10 & 13 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} 5 & 1 & 6 & 9 & 12 \\ 2 & 7 & 11 & 8 & 14 \\ 3 & 4 & 10 & 13 & \infty \end{bmatrix} . \qquad (2.9)$$

Notar el efecto del “relleno” ∞ hecho con el argumento opcional `pad`. Como `any (...)` sigue siendo `true` entonces sigue iterando el lazo `while` más interno

$$t = \begin{bmatrix} 5 & 1 & 6 & 9 & 12 \\ 2 & 7 & 11 & 8 & 14 \\ 3 & 4 & 10 & 13 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 5 & 6 & 12 & 9 \\ 2 & 7 & 8 & 11 & 14 \\ 3 & 4 & 10 & 13 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 5 & 6 & 9 & 12 \\ 2 & 7 & 8 & 11 & 14 \\ 3 & 4 & 10 & 13 & \infty \end{bmatrix} ; \tag{2.10}$$

y ahora sale del lazo `while` más interno. En la última pasada tendremos

$$\begin{aligned} t &= \begin{bmatrix} 1 & 2 & 3 & 5 & 7 & 4 & 6 & 8 & 10 & 9 & 11 & 13 & 12 & 14 \end{bmatrix} \\ &\rightarrow \begin{bmatrix} 1 & 2 & 3 & 5 & 4 & 7 & 6 & 8 & 9 & 10 & 11 & 13 & 12 & 14 \end{bmatrix} \\ &\rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \end{bmatrix} . \end{aligned} \tag{2.11}$$

2.6. Dispersión de índices disponible desde F90

Por ejemplo, consideremos los arreglos

$$\begin{aligned} a &= \begin{bmatrix} A & B & C & D & E & F \end{bmatrix} ; \\ i &= \begin{bmatrix} 1 & 3 & 5 & 2 & 4 & 6 \end{bmatrix} ; \\ j &= \begin{bmatrix} 3 & 2 & 3 & 2 & 1 & 1 \end{bmatrix} ; \end{aligned} \tag{2.12}$$

con $n = |a| = |i| = |j| = 6$, con $1 \leq i_s, j_s \leq 6$ y $1 \leq s \leq 6$, en donde i no tiene índices repetidos pero j si. Bajo tales condiciones, con F95 sólo son posibles instrucciones de dispersión de índices de la forma

```
b = a (i)
c = a (j)
d (i) = a
```

Estas asignaciones producen los siguientes movimientos

$$\begin{aligned} b &= \begin{bmatrix} a(i_1) & a(i_2) & a(i_3) & a(i_4) & a(i_5) & a(i_6) \end{bmatrix} = \begin{bmatrix} a_1 & a_3 & a_5 & a_2 & a_4 & a_6 \end{bmatrix} ; \\ c &= \begin{bmatrix} a(j_1) & a(j_2) & a(j_3) & a(j_4) & a(j_5) & a(j_6) \end{bmatrix} = \begin{bmatrix} a_3 & a_2 & a_3 & a_2 & a_1 & a_1 \end{bmatrix} ; \\ d(i) &= \begin{bmatrix} d(i_1) & d(i_2) & d(i_3) & d(i_4) & d(i_5) & d(i_6) \end{bmatrix} = \begin{bmatrix} d_1 & d_3 & d_5 & d_2 & d_4 & d_6 \end{bmatrix} ; \end{aligned} \tag{2.13}$$

por lo que la tercera asignación de índices, `d(i)= a`, equivale a

$$\begin{bmatrix} d_1 & d_3 & d_5 & d_2 & d_4 & d_6 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 \end{bmatrix} ; \tag{2.14}$$

reordenando

$$\begin{bmatrix} d_1 & d_2 & d_3 & d_4 & d_5 & d_6 \end{bmatrix} = \begin{bmatrix} a_1 & a_4 & a_2 & a_5 & a_3 & a_6 \end{bmatrix} ; \tag{2.15}$$

en definitiva,

$$\begin{aligned} b &= \left[\begin{array}{cccccc} A & C & E & B & D & F \end{array} \right] ; \\ c &= \left[\begin{array}{cccccc} C & B & C & B & A & A \end{array} \right] ; \\ d &= \left[\begin{array}{cccccc} A & D & B & E & C & F \end{array} \right] . \end{aligned} \tag{2.16}$$

El programa demo `scatter1` muestra este comportamiento, en donde además usamos la *extensión* aprobada en el F03 de usar corchetes `[..]` para definir un arreglo,

```
program scatter1
  implicit none
  integer, dimension (6) :: b, c, d
  integer, dimension (6) :: a = [ 1, 2, 3, 4, 5, 6 ]
  integer, dimension (6) :: i = [ 1, 3, 5, 2, 4, 6 ]
  integer, dimension (6) :: j = [ 3, 2, 3, 2, 1, 1 ]
  b = a (i)
  c = a (j)
  d (i) = a
  !notar que d (j) = c  es ilegal para estos indices "j"
  write (*,*) "Arreglos, scatter de indices en F90/F95"
  write (*,100)" i         : ", i
  write (*,100)" j         : ", j
  write (*,100)" a         : ", a
  write (*,100)" b = a (i) : ", b
  write (*,100)" c = a (j) : ", c
  write (*,100)" d (i) = a : ", d
  100 format (a, 10 (1x,i6))
end program
```

pero `d(j)=c` es ilegal (ejercicio: por qué ?).

2.7. Diagonal principal de una matriz en formato ralo

El siguiente ejemplo extrae la diagonal principal de una matriz almacenada en el formato ralo dado por el triplete (i, j, a_{ij}), usando la intrínseca `pack` y una máscara booleana.

```
program  p12
  implicit none
  integer, dimension (11) :: ii = [ 5,2,1,3,4,3,5,4,3,2,1 ]
  integer, dimension (11) :: jj = [ 5,5,1,4,2,3,3,4,1,2,4 ]
  integer, dimension (11) :: aa = [ 1,2,3,4,5,6,7,8,9,1,2 ]
  logical, dimension (11) :: mask
  integer, dimension (:), allocatable :: i
  integer :: k, n, p
  character (64) :: s
  write (*,*) "Extrae diagonal principal de una matriz en formato ralo"
  n = maxval (ii)
  p = size (ii)
  write (*,*) "nro de incognitas         ; n = ", n
  write (*,*) "nro de coeficientes ralos ; p = ", p
  allocate ( i (n) )
  mask = ( ii (1:p) == jj (1:p) )
  write (*,*) "fila matriz rala ; ii   = ", ii
  write (*,*) "colu matriz rala ; jj   = ", jj
  write (*,*) "mascara          ; mask = ", mask
  k = count (mask)
  write (*,*) "nro de elementos en la diag ppal hallados ; k = ", k
  i (1:k) = pack (ii (1:p), mask)
  s = "indices de la diagonal principal extraidos de la matriz rala"
  write (*,100) s ; write (*,*) i
  100 format (1x, a,10(1x,i3))
end program
```

Capítulo 3

Extensión HPF2 del Fortran

3.1. Paralelismo en los datos

El *paralelismo en los datos* se refiere a la simultaneidad de cálculo, o concurrencia, que se logra cuando una misma operación se puede hacer simultáneamente con algunos o todos los elementos de un arreglo de datos. Un programa con paralelismo en los datos implementa una sucesión de operaciones simultáneas a través de técnicas de descomposición en subdominios a las estructuras de datos. Un paradigma de programación con datos en paralelo es:

- de *inter-comunicación implícita*, pues al estar a un nivel abstracto mayor, no se le exige al programador explicitar las estructuras de inter-comunicación y de coherencia de los datos distribuidos;
- relativamente más restrictivo, pues no todos los algoritmos paralelos pueden implementarse en términos de paralelismo en los datos.

Por eso, aunque útil, no es un paradigma universal de programación en paralelo. Aquí sólo nos concentramos en un *particionamiento de los datos*, mediante construcciones explícitas e implícitas, tales que mejoren la simultaneidad y el balance de carga. Lo interesante de este paradigma es que el programador *sugiere* al compilador, mediante directivas, cómo debería distribuir y alinear los arreglos sobre los procesadores, mientras que el compilador se encarga de definir todas las operaciones de comunicación. Típicamente, el programador es el responsable en sugerir el modo de la descomposición empleado para el dominio, mientras que el compilador se encargará de los detalles y de la comunicación involucrada, rechazando la distribución si el costo fuera muy alto. Un objetivo en cálculo paralelo busca asignar a todos los procesadores, aproximadamente, la misma cuota de trabajo, o balance de carga, y con la más baja la comunicación posible entre procesadores.

3.2. Máquinas paralelas SSP y MMP

Siguiendo a Press *et al.* [34] distinguiremos:

- *Máquinas Paralelas de Pequeña Escala* [Small-Scale Parallel machines (SSP)]: cuando el número de procesadores p disponibles es mucho menor que el tamaño típico del problema n, i.e. $p \ll n$, por ejemplo los *cluster* Gerónimo y Aquiles del CIMEC;

- *Máquinas Masivamente Paralelas* [Massively Multiprocesador machines (MMP)]: cuando el número de procesadores p disponibles es mucho mayor que n, i.e. $p \gg n$ o, al menos, $p \approx O(n^2)$. En este caso la memoria RAM disponible será usualmente enorme.

La razón de la distinción entre equipos SSP y MMP argüida por Press *et al.* [34], se debe a que se pueden concebir algoritmos paralelos que sean razonablemente prácticos sobre las MMP pero inviables sobre una SSP. Por ejemplo, en la factorización LU en paralelo del *Numerical Recipes* se emplea reiteradamente el producto *exterior* de vectores temporarios mediante la intrínseca `spread` del F90, conduce a crear arreglos temporarios *on the fly* de un tamaño comparable al de la matriz a triangularizar, lo cual puede agotar rápidamente la memoria RAM disponible. Así, algoritmos paralelos orientados o bien a una máquina SSP o bien a una MMP pueden ser muy diferentes en cuanto a su concepción y practicidad.

3.3. Algunos principios de diseño del HPF

El HPF es un conjunto de extensiones normalizadas para hacer un cierto tipo de paralelismo restringido en los lenguajes F77 y F90/F95 (omitimos el F03 pues es muy reciente y no está contemplado todavía por HPF). Asume el paradigma de programación en paralelo basado en un *paralelismo en los datos*, en el cual

- se asume que existe algún conjunto de procesadores, ya sea como un multiprocesador, esto es, una única computadora con varios procesadores encapsulados, o bien como un multicomputador, por ejemplo, un cluster);
- cada procesador ejecuta el mismo código, es decir, se asume un modelo *Single Program, Multiple Data* (SPMD);
- algunos de los arreglos multi-índices estarán distribuidos sobre las memorias asgnadas a cada procesador, mientras que los remanentes estarán *replicados*;
- cada procesador opera sobre una parte de los datos distribuídos;
- en forma oculta al programador se hacen pasos de mensajes entre los procesadores, por lo que no hace falta programar el envío o recepción de los paquetes;
- se concede una *baja* sincronización, esto es, los procesadores no necesariamente hacen exactamente la misma tarea en el mismo ciclo de tiempo. Sólo se impone la sincronización en ciertas partes críticas del programa y que no es controlado por el programador.

En la práctica, en un programa fuente secuencial Fortran:

- Se introducen ciertas *directivas declarativas* en donde *sugerimos* al compilador una cierta *distribución* y *alineamiento* relativo de los datos matriciales sobre un arreglo abstracto de procesadores, ya sea en forma explícita (e.g. declaramos un arreglo lineal de 4 × 2 procesadores) o bien implícito (o sea, no declaramos nada y confiamos en lo que decida hacer el compilador);

- Se agregan *directivas ejecutables* en donde se proponen zonas del código que deberían efectuarse en paralelo (sobre los datos matriciales distribuidos);

- Todas las directivas al compilador, como es frecuente en las extensiones de lenguaje, se intercalan como comentarios especiales en los lugares apropiados, de modo tal que serán ignoradas por

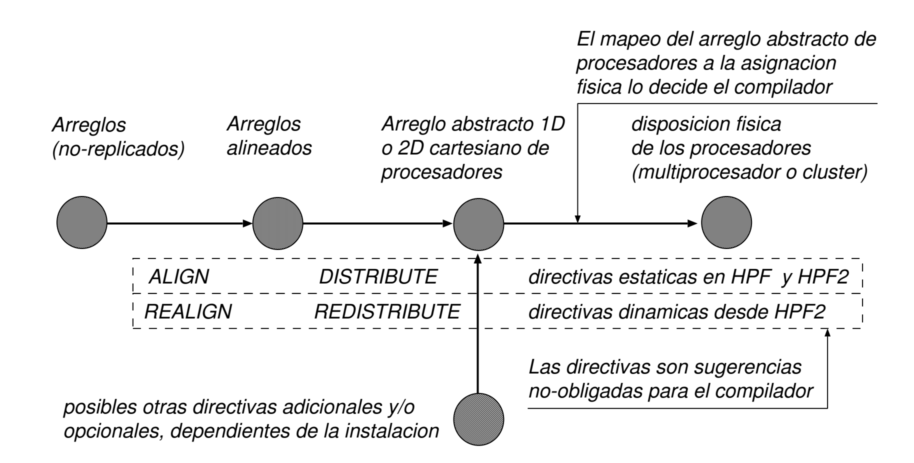


Figura 3.1: Etapas en la asignación de los datos distribuidos a los procesadores.

un compilador secuencial Fortran. Es decir que las mismas empezarán con el símbolo especial reservado para indicar un comentario, ya sea `c` en F77 o bien ! en F90;

- Algunas de las *directivas* HPF especifican la distribución de los datos, sugiriendo además a los procesadores cómo acceder a los mismos;
- El lenguaje de programación F90-F95 con su sintaxis para arreglos multi-índice y su librería intrínseca, ya prevee el cálculo en paralelo. Esto quiere decir, por ejemplo, que la intrínseca `matmul (a,b)` evaluará el producto matricial en forma secuencial cuando usemos un compilador F90-F95 sobre un monoprocesador pero, si los arreglos `a` y `b` estuvieran distribuidos y usamos un compilador HPF, entonces la `matmul` hará el cálculo en forma distribuida. Ademas el HPF extiende la librería intrínseca e introduce la *directiva ejecutable* `independent`.

Como todas las directivas HPF son consideradas como un comentario para un compilador secuencial F90/F95, un programa HPF podrá ser compilado sin alteración por un compilador F90/F95. También podría hacerse al revés, esto es, un programa en F90/F95 "puro"podría ser compilado por un compilador HPF y ser paralelizado por omisión según los recursos de parlalelismo disponibles por el *hardware* del sistema, aunque muy probablemente no será al forma más efectiva de paralelizar un código secuencial dado. En la Fig. 3.1 se resume las etapas en la asignación de los datos distribuidos a los procesadores. Notar que hay dos de estas etapas: una "abstracta" que es responsabilidad del programador, y otra "física", que es responsabilidad del compilador.

3.4. HPF1x y el paralelismo en los datos

Al escribir un programa HPF hay que:

- Proponer una grilla lineal abstracta de procesadores, la cual no necesita tener relación con la configuración física del equipo. Incluso, muchas veces, omitimos esta tarea y la delegamos al compilador;

- Proponer una distribución de los datos sobre el arreglo abstracto de los procesadores;
- Intentar manipular los datos distribuidos en forma conveniente mediante el uso de:
  - instrucciones en paralelo implícitas por usar sintaxis matricial F95/F90 y librería intrínseca;
  - emplear el estamento `forall` toda vez que sea posible. Recordar que es un lazo `do` generalizado en donde el orden de cómputo no necesariamente es el secuencial;
  - para que los procedimientos y funciones puedan ser llamados dentro de un lazo en paralelo hay que darles el atributo `pure`, lo cual significa que son *libres* de "efectos colaterales", esto es, las diversas instancias de los procedimientos/funciones puros que son llamados en paralelo no interfieren entre sí;
  - ocasionalmente puede ser de interés emplear la `extrinsics`, esto es, procedimientos que fueron escritos en un lenguaje diferente al F90-F95 (e.g. en C);
  - aunque conspira contra la *portabilidad* del código, podría ser útil emplear la `hpf_library` con otras intrísecas no-disponibles en F95, e.g. ordenar un arreglo en paralelo con la `grade_up` o usar las `xxx_scatter` para operaciones en paralelo que involucren dispersión de índices (e.g. producto matriz-rala vector).

3.5. Declaraciones y directivas HPF

- Las declaraciones y directivas HPF son de la forma `!hpf $<hpf-declaracion>` y `!hpf $<hpf-directiva>`, respectivamente. Por su definición representan comentarios para un compilador secuencial F90/F95 pero son tenidas en cuenta por un compilador o traductor HPF. En ningún caso cambiarán la *semántica* del código secuencial (qué es lo que se quiere hacer), sólo se le sugiere en *dónde* lo tiene que hacer, por lo que los resultados alcanzados con los códigos sequencial y paralelo deberían ser los mismos;

- Las declaraciones y directivas son *sugerencias* al compilador HPF, o sea, que las puede modificar o, frecuentemente, ignorar;

- Todos los identificadores utilizados en las declaraciones y directivas HPF deben ser diferentes de los identificadores usados en el resto del programa, es decir, debe existir un *único* espacio global de los identificadores de usuario;

- La mayoría de las directivas *declarativas* se refieren a una distribución *pretendida* en los datos. La única directiva *ejecutable* es la `independent`, la cual dispone de algunas *cláusulas* opcionales que modifican su comportamiento, tales como `new` y `reduction`;

- Es posible escribir las directivas HPF ya sea con los estilos F77 o F90 pero, al usar el `&` como indicador de continuación en F90, hay que empezar la nueva línea con la directiva de inicio `!hpf$`. En estas notas usaremos casi siempre el estilo F90.

3.5.1. Directivas declarativas

Son de la forma, por ejemplo,

```
!hpf$ processors, dimension (n)     :: p      !arreglo 1D de procesadores
!hpf$ template, dimension (n,n)     :: t1, t2 !plantillas 2D
!hpf$ distribute (cyclic)           :: a      !mapeo ciclico 1D
!hpf$ distribute (block)            :: b      !mapeo bloques 1D
!hpf$ distribute (cyclic,*)         :: c      !colapso en indice 2
!hpf$ distribute (cyclic,*) onto p :: d      !mapeo sobre p
```

3.5.2. Directivas ejecutables

Típicamente definen si un lazo `do` o `forall` puede o no hacerse en paralelo. En tal caso, a veces, habrá que hacer copias locales de variables auxiliares usando la cláusula `new`, o bien hacer operaciones de reducción usando la `reduction`. Por ejemplo,

```
!hpf$ independent, new (i)        !variable local en cada nodo
!hpf$ independent, reduction (s)  !operacion de reduccion sobre s
```

Otro ejemplo, si nos "olvidamos" de la intrínseca `dot_product` disponible desde F90 para el producto escalar de los vectores `a` y `b`, alternativamente podríamos codificarlo como:

```
s = 0.0
!hpf$ independent, reduction (s)  !reduccion binaria
do  i = 1, n
  s = s + a (i) + b (i)            !producto escalar
end do
```

3.6. Directiva declarativa del arreglo de procesadores abstracto

Define una grilla abstracta de procesadores, o *nodos*, en R^n, la cual no tiene relación con el *hardware* subyacente. Se asume la topología cartesiana simple, es decir, es decir, no prevee ni arreglos circulares (*toros*) ni topologías más elaboradas (e.g. *hiper-cubos*). Dado que su uso es opcional, puede y suele omitirse pero, en tal caso, lo decide el compilador. Por ejemplo,

```
!hpf$ processors                       :: p4 !arreglo 0D de nodos
!hpf$ processors, dimension (n)        :: p1 !arreglo 1D de nodos
!hpf$ processors, dimension (n,m)      :: p2 !arreglo 2D de nodos
!hpf$ processors, dimension (p,q,r) :: p3 !arreglo 3D de nodos
```

3.7. Directiva declarativa para la distribución de los datos

Las directivas para distribuir los datos en paralelo, por ejemplo, son de la forma

```
real, dimension (100)    :: a
real, dimension (10,10) :: b, c, d, e
!hpf$ distribute (block)         onto p1 :: a     !distribucion 1D
!hpf$ distribute (cyclic,block) onto p2 :: b, c !distribucion 2D
!hpf$ distribute (cyclic,*)      onto p1 :: d     !colapso indice 2
!hpf$ distribute e (cyclic,*) onto p1             !otra sintaxis
```

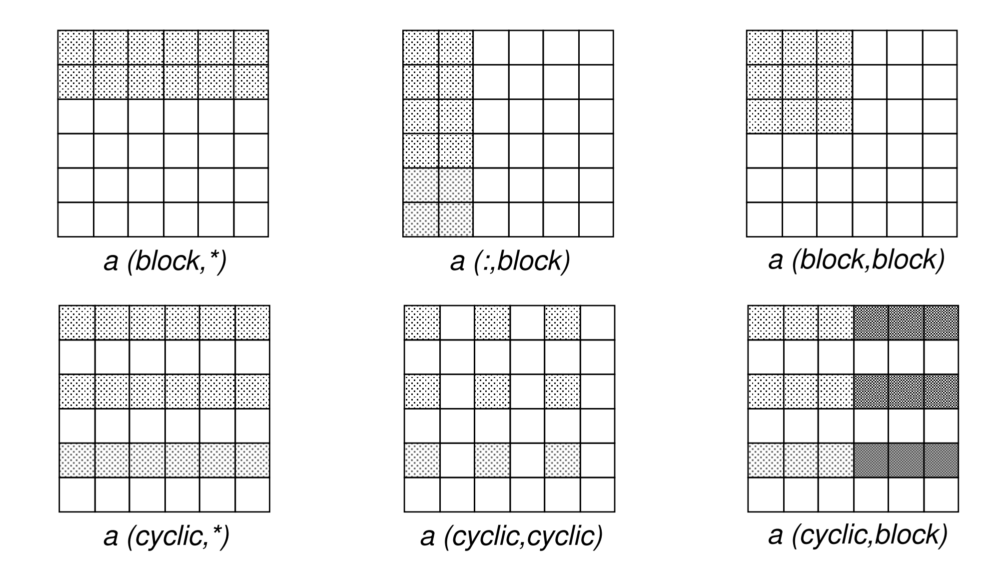


Figura 3.2: Ejemplos de la directiva `distribute` en arreglos de dos índices.

- La directiva `distribute` sugiere al compilador cómo distribuir los datos entre procesadores. En el HPF1 se dispone sólo de dos tipos básicos de mapeo, dados por el `block` y el `cyclic`, mientras que en HPF2 se agregaron otras algo más flexibles (luego mencionadas);
- La distribución `block` divide al arreglo `a` en bloques o "porciones"(*chunks*) contiguos, de igual tamaño excepto, posiblemente, el último que puede ser menor, y asigna cada bloque a un procesador. Es útil en cálculos que involucren elementos vecinos (e.g. en FDM);
- En el HPF1 sólo es posible controlar el tamaño con `block(h)` o `cyclic (h)`. Pero tiene que ser *uniforme*, lo cual conspira para datos con distribución irregular. En cambio, en el HPF2 es posible dar un tamaño variable;
- La distribución `cyclic` distribuye los elementos del arreglo `d` en el mismo modo en que son distribuidas las cartas: cada procesador recibe un elemento por vez y, sólo después que todos hayan recibido un elemento, hace una segunda pasada y así hasta terminar. En general ayuda a un mejor balance de carga para tareas muy acopladas ;
- Poner un `*` en algún índice significa que la distribución es ignorada en ese índice, esto es, todos el rango de elementos del índice que tenga un `*` va a parar al mismo procesador;
- Estas directivas pueden combinarse en forma arbitraria en cada índice de los arreglos distribuidos. Por ejemplo, en la Fig. 3.2, se muestran algunas posibilidades en 2D;
- Si se omite la cáusula `onto`, entonces los objetos son distribuidos en la grilla por omisión, la cual puede darse como un argumento en la línea de comandos al lanzar la corrida;
- Todo esto vale también para la distribución de arreglos dinámicos `allocatables`;
- Todas las variables escalares serán *replicadas* en cada procesador.

Para ver mejor cómo se mapea por bloques y cíclicamente consideremos los arreglos de un índice `a` y `b` dados por el fragmento de código

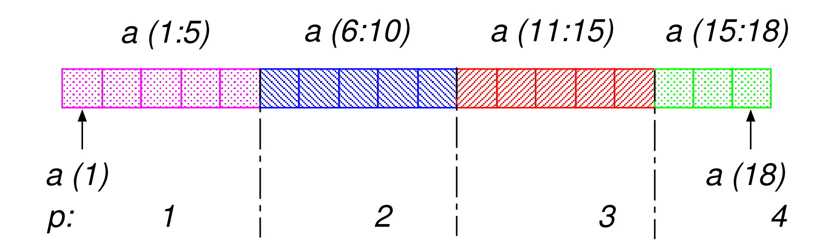


Figura 3.3: Ejemplos de la directiva `distribute block` en el arreglo `a` de un índice.

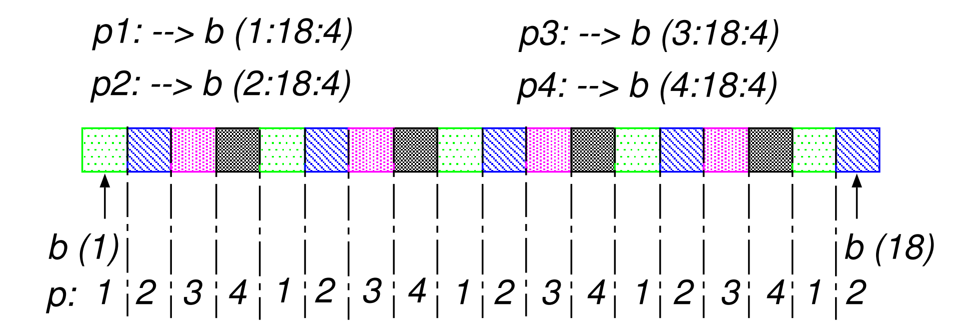


Figura 3.4: Ejemplos de la directiva `distribute cyclic` en el arreglo `b` de un índice.

```
real, dimension (18) :: a, b
!hpf$ processors, dimension (4)   :: p   !arreglo de nodos 1D
!hpf$ distribute (block)  onto p :: a   !bloques 1D
!hpf$ distribute (cyclic) onto p :: b   !ciclico 1D
```

En este caso (ver Figs. 3.3-3.4):

- En la distribución `block`, como `size (A) = 18` y `size (P1) = 4`, cada uno de los 4 procesadores recibe $z = \lceil 18/4 \rceil = 5$ elementos, excepto el último que se queda con el remanente $y = 18 - 3 \times 5 = 3$;
- En cambio, en la distribución `cyclic`, cada uno de los cuatro procesadores recibe un elemento por vez y cuando todos recibieron un elemento hace una segunda pasada y así hasta cubrir todo el arreglo.

A la hora de distribuir los arreglos sueler ser conveniente:

- usar sintaxis matricial y las intrínsecas matriciales;
- evitar re-mapeos;
- cuanto mayor es el número de procesadores, mayor será la comunicación y menor el rendimiento;
- al intentar mejorar el balance de carga probar con las reglas del LHS y del RHS discutidas en la Sec. 3.12;
- tratar de asegurar la mayor localidad de los datos posible;
- escribir primero una versión serial correcta cuya algoritmia prevea, en cierta medida, algún modelo de paralelismo y testearla;

- luego agregar directivas tratando de reducir el número de comunicaciones y compilar en forma interactiva.

3.8. Replicación de las variables escalares

Por omisión las variables escalares son "replicadas", i.e. cada procesador dispone de una copia de las mismas. El compilador HPF debe asegurarse de que sea un valor coherente. Por ejemplo, en

```
  integer, parameter :: n = 1024
  integer, dimension (n,n) :: a
  integer :: s
  !hpf$ distribute (block, block) :: a
  i = indice_i (a) ; j = indice_j (a)
  s = a (i,j)
```

el procesador que disponga del elemento `a(i,j)` actualizará su copia de `s` y enviará el nuevo valor a los demás procesadores mediante un *broadcast*. Si se ncesita un escalar distinto en cada procesador hay que replicarlo en cada uno usando un `new`, o bien hay que declarar un arreglo auxiliar `s` con tantos elementos como procesadores haya.

3.9. Directiva declarativa de alineamiento

Los alineamientos relativos entre dos o más arreglos aseguran que los elementos correspondientes residan en el mismo procesador. La idea es usar patrones de distribución para los arreglos y decidir cómo deben ser alineados en forma relativa según la tarea en paralelo que se dese hacer. Siempre es conveniente efectuar una alineación, ya que puede:

- reducir la comunicación;
- mejorar el balance de carga;
- mejorar la localidad de los datos.

Existen varias opciones en cuanto a su sintaxis y las veremos mediante algunos ejemplos. Notar en todos ellos que debe haber un balance en los índices distribuidos y en los colapsados a ambos lados de la palabra reservada `with`, que forma parte de la instrucción `align`, como puede verse también en la Fig. 3.5.

3.9.1. Alineamiento 1D simple

Por ejemplo, para alinear los elementos a_i, b_i de los arreglos `a`, `b` con respecto al elemento t_i de otro arreglo `t` que fue distribuido por bloques, usamos la directiva declarativa `align`. Pero, en el caso especial de `ADAPTOR`, hay que distinguir si usamos alocamiento *estático*

```
program alinea_estatico
  implicit none
  integer, parameter :: n = 1024
  integer, dimension (n) :: a, b, t
  !hpf$ distribute (block) :: t
  !hpf$ align (:) with t (:) :: a, b
end program
```

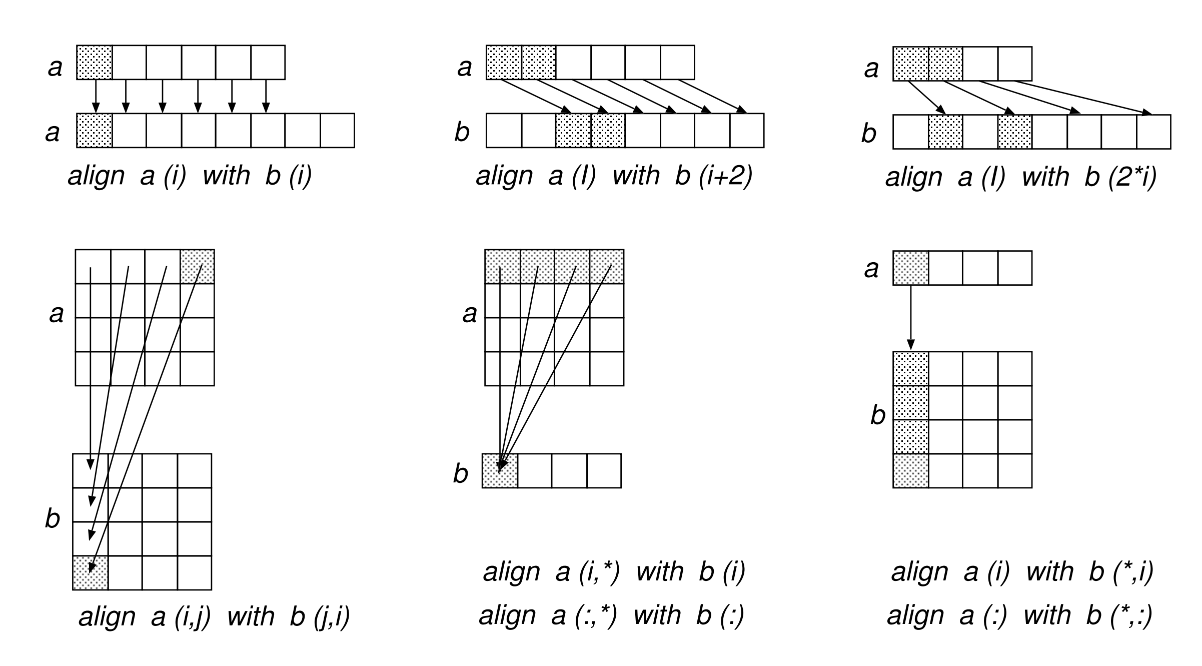


Figura 3.5: Ejemplos de la directiva `align` en HPF.

o alocamiento *dinámico*

```
program alinea_dinamico
  implicit none
  integer :: i
  integer, dimension (:), allocatable :: a, b, t
  !hpf$ distribute (block) :: t
  !hpf$ align (i) with t (i) :: a, b
  write (*,*)" n ? " ; read (*,*) n
  allocate (a(n),b(n),t(n))
end program
```

en donde, por restricciones de ese traductor, se requiere usar la notación indicial a través del índice entero i que debe ser declarado antes del `align`.

3.9.2. Alineamiento 2D simple

Por ejemplo, para alinear los elementos a_{ij}, b_{ij} de los arreglos `a`, `b`, con respecto al elemento t_{ij} de otro arreglo `t` que fue distribuido por bloques en ambos índices, podemos escribir

```
  integer, parameter :: n = 1024
  integer, dimension (n,n) :: a, b, c, t
  !hpf$ distribute (block,block) :: t
  !hpf$ align (:,:) with t (:,:) :: a, b, c
```

Este alineamiento es conveniente, entre otros casos, para operaciones matriciales de la forma

```
  c = a + b + a * b   !ya que todo es local
```

3.9.3. Alineamiento 2D traspuesto

A veces es mejor usar índices para destacar mejor cómo se debe hacer el alineamiento. En ejemplo anterior, si ahora queremos que `b` quede traspuesta con respecto a `a` y ésta alineada con respecto a `t` podríamos poner

```
implicit none
integer, parameter :: n = 1024
integer, dimension (n,n) :: a, b, c, t
integer :: i, j
!hpf$ distribute (block,block) :: t
!hpf$ align (i,j) with t (:,:) :: a
!hpf$ align (j,i) with t (:,:) :: b
```

lo cual es conveniente en operaciones matriciales que involucren matrices traspuestas, pues la sentencia

```
c = a + transpose (b)     !todo es local
```

se hace en paralelo sin comunicación. Notar que los índices `i`, `j` sólo son "símbolos" que se usan para destacar cuáles son las dimensiones que deben alinearse, y por ello no interesan sus valores iniciales ni finales.

3.9.4. Alineamiento con incremento no unitario

Por ejemplo, para alinear los elementos a_i, b_{2i} podemos hacer

```
implicit none
integer, parameter :: m = 5, n = 10
integer, dimension (m) :: a
integer, dimension (n) :: b
!hpf$ distribute (block,block) :: b
!hpf$ align a (i) with b (2*i-1)
!hpf$ align a (:) with b (1:n:2) !sintaxis alternativa
```

Este alineamiento es apropiado en operaciones con arreglos tales como

```
a = a + b (1:n:2)         !ya que todo es local
```

3.9.5. Alineamiento con incremento en reversa

Por ejemplo, para alinear cada elemento de `a` con cada elemento par de `b` podemos hacer

```
implicit none
integer, parameter :: m = 5, n = 10
integer, dimension (m) :: a
integer, dimension (n) :: b
!hpf$ distribute (block,block) :: b
!hpf$ align a (i) with b (n-2*i+2)
!hpf$ align a (:) with b (n:1:-2)      !sintaxis alternativa
```

Este alineamiento es conveniente para operaciones matriciales del tipo

```
a = a + b (n:1:-2)        !ya que todo es local
```

3.9.6. Alineamiento con replicación en algún índice

Un caso usual es aquel en que un mismo intervalo de un arreglo se usa en cada nodo. Por ejemplo, podemos alinear elementos en la forma

```
integer, parameter :: n = 8
integer, dimension (n,n) :: a
!hpf$ distribute (block,block) :: a
!hpf$ align y (:) with a (*,:)
!hpf$ align y (j) with a (*,j)  !sintaxis alternativa
```

Aquí la dimensión en donde figura el símbolo `*` es "colapsada". En este caso, una copia de cada elemento `y(j)` es alineada con cada columna `a(:,j)`, para $1 \le j \le n$. Entonces, cada nodo que reciba un elemento `a(:,j)` recibirá también una copia del elemento `y(j)`.

3.10. Directiva declarativa de una plantilla (`template`)

NO tiene nada que ver con los *templates* disponibles en C++ !! En HPF son sólo mapas de índices que no ocupan memoria y deben ser definidos en forma estática. Una forma de pensar un *template* en HPF es imaginarlo como un arreglo de elementos donde cada uno ocupa 0 *bytes*. Pueden emplearse para hacer los alineamientos en forma algo más concisa. Son entidades que deben tener un identificador y no pueden transferirse entre procedimientos. De todos modos, hay que destacar que hay quienes las cuestionan argumentando que son entes innecesarios en la extensión.

Las plantillas deben ser primero declaradas, a continuación distribuidas y, finalmente, usadas, esto es, los demás arreglos pueden ser alineados con respecto a éstas. Las formas y reglas de distribución son las mismas que las vistas para arreglos. Por ejemplo, reconsideremos el programa visto en la Sec. 3.9.1 re-escribiéndolo como

```
integer, parameter :: n = 1024
!hpf$ template, dimension (n) :: t
!hpf$ distribute (block) :: t
integer, dimension (n) :: a, b
!hpf$ align (:) with t (:) :: a, b
```

3.11. Normas HPF1 y HPF2

Lo visto hasta ahora era lo previsto en las normas HPF1x. Pero la experiencia mostró ciertas deficiencias. Entre las más notables se destacan:

- sólo se puede hacer paralelismo con datos distribuidos en forma regular mediante las directivas de distribución por bloques `block` y `block(n)`, o bien en forma cíclica con `cyclic` y `cyclic (n)`;
- no se prevee el paradigma del paralelismo en las tareas;
- falta de opciones para inter-operabilidad con otros paradigmas de paralelismo.

Por eso, su última revisión dio lugar a la norma HPF2 que es soportada por los traductores de libre distribución `ADAPTOR`[5] y `FHPF`[22] y, recientemente, por el compilador comercial `Portland PGI`[33]. Por ejemplo, en el *HPF language Reference Manual* del traductor ADAPTOR que está en el directorio

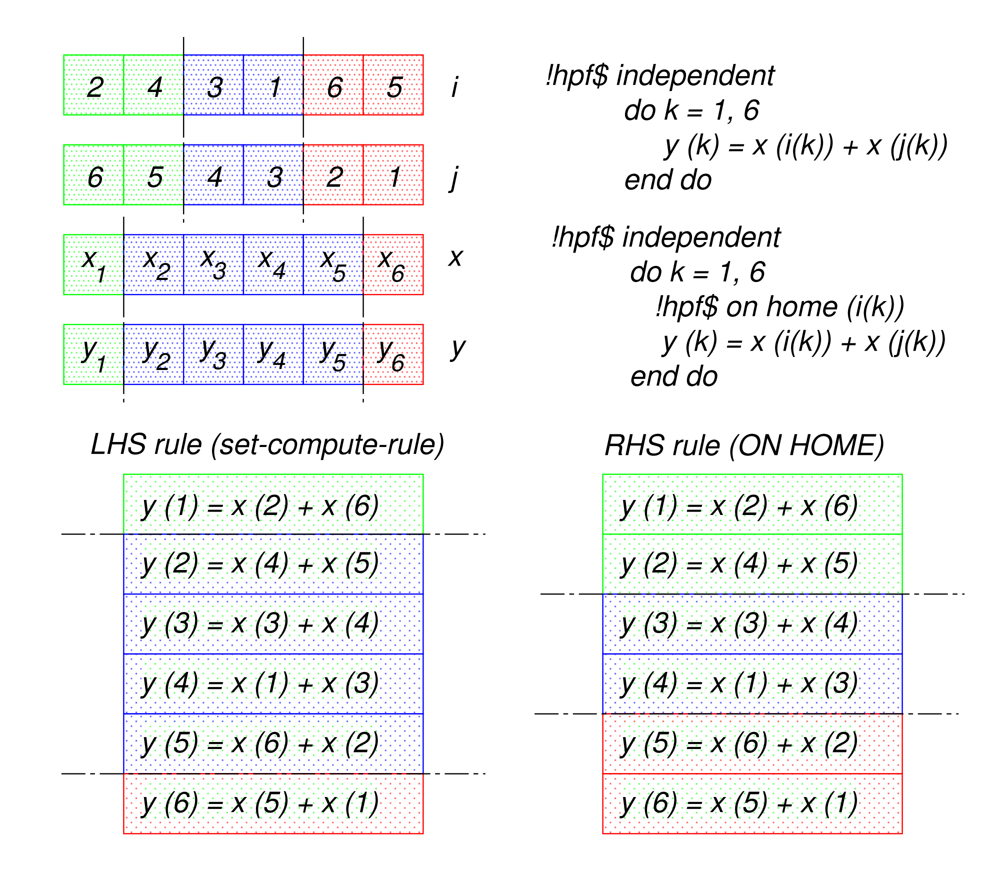


Figura 3.6: La regla del LHS por omisión y la regla del RHS mediante la cláusula optativa `ON HOME`.

doc de la instalación se analizan, entre otros cambios introducidos a partir de HPF2, las siguientes mejoras:

- las estrategias en el *mapeo de datos* que son presentadas en el Cap. 5, pp. 16-25. En particular, las distribuciones `gen_block (siza)`, `indirect (mapa)` y `arbitrary (n,siza,mapa)`, discutidas en la Sec. 5.2, pp. 17-19;
- las estrategias en la *distribución de datos* del Cap. 6, pp. 26-30. En particular, la directiva `shadow`;
- el *paralelismo de datos* que son presentadas en el Cap. 9, pp. 40-43. En particular, la inclusión de las cláusulas `on_home` y `resident`;
- el *paralelismo de tareas* con la directiva `task_region` y la `hpf_task_library`, ver el Cap. 10, pp. 44-45;
- los procedimientos extrínsecos `hpf_local`, `hpf_serial`, `f77_local` y `f77_serial`, ver el Cap. 11, pp. 46-49

3.12. La dos reglas básicas en las asignaciones en HPF2

En el paradigma de paralelismo en los datos con HPF2.x, las operaciones de cálculo sobre los datos distribuidos son controladas por dos reglas (ver la Fig. 3.6):

Regla del LHS: por omisión las operaciones son realizadas por el procesador que tenga asignado el LHS (set-compute-rule) : en una operación de cálculo y asignación que involucre arreglos distribuidos, por omisión, el procesador que tenga asignado el elemento que aparezca en el miembro *izquierdo* (LHS) de la instrucción, realizará las operaciones aritméticas necesarias y la posterior asignación del resultado. Por lo tanto, será el responsable de conseguir todos los datos necesarios del miembro derecho (RHS), efectuar todas las cuentas y hacer la asignación, intercalando las operaciones de inter-comunicación necesarias.

Regla del RHS: la cláusula optativa ON HOME (X) prescribe que las operaciones serán realizadas por el procesador que tenga asignado el arreglo distribuido `X` del RHS : modifica la regla del LHS seguida por omisión, indicando ahora que el responsable de las operaciones será el procesador que tenga asignado el elemento i de un arreglo distribuido `X` que aparece en el miembro *derecho* (RHS) de la instrucción.
Empero, no siempre estas reglas son respetadas por un compilador HPF. Por ejemplo, si los componentes derechos de la expresión estuvieran todos en un mismo procesador, entonces la cuenta la hará el procesador que disponga del RHS y el resultado a asignar será enviado al procesador que tenga el LHS. El criterio básico que emplea un compilador HPF es el de reducir todo lo posible el número de comunicaciones.

3.13. Cálculo en paralelo con datos distribuidos en HPF

Las directivas HPF para distribuir los datos sobre los nodos sólo indican al compilador HPF cómo debería distribuir y alinear los arreglos, pero no expresan instrucciones de cálculo en paralelo. Para esto último hay que recurrir al uso:

- de la sintaxis mult-índice del F90-F95;
- de la librería mult-índice intrínseca del F90-F95;
- del estamento `forall` del F95;
- de la directiva HPF *independent*;
- de variables con atributo `new` el cual permite definir variables locales dentro de un lazo paralelo;
- de las operaciones de reducción dentro de un lazo paralelo con `reduction`;
- del atributo `pure` para las funciones y subrutinas dentro de un lazo que se ejecuta en paralelo, el cual, simplemente, garantiza que las llamadas estarán libres de *efectos colaterales* indeseables. Notar que *todas* las intrínsecas F95 son `puras`.

3.13.1. Instrucción `forall`

La instrucción `forall` es estándar a partir de F95. Permite expresar asignamientos simultáneos en forma concisa y versátil, admite máscaras booleanas, usando un `where`, y puede invocar funciones y procedimientos que tengan el atributo `pure`. En general, su mecanismo de funcionamiento es diferente de la instrucción `do` (ver Sec. 2.2). A continuación, veamos algunos ejemplos:

- Acceder a posiciones que no son inmediatas para un lenguaje vectorizado. Como un primer ejemplo,

```
forall (i=1,n) a (i,i) = b (i)
```

asigna el vector `b` en la diagonal principal de la matriz `a`. Y si se quiere asignar un vector `b` a la parte triangular inferior inclusive la diagonal principal podríamos hacer

```
forall (i=1,j)
  forall (i=1,j) a (i,j) = b (i)
end forall
```

Como alternativa podría ponerse

```
!hpf$ independent, new (i)
do j = 1, n
  forall (i=1,j) a (i,j) = b (i) !triangular inferior
end do
```

en donde el lazo `do` es paralelizado mediante la directiva `independent` y se le agrega el atributo `new` para tener una variable local (*replicada*) en cada procesador;

- Usar índices en el RHS, e.g.

```
forall (i=1,n, j=1:n, j.ne.i) a (i,j) = i+j !fuera de la diagonal
```

- Llamar funciones y procedimientos que tengan el atributo `pure`, e.g.

```
forall (i=1,n, j=1:n) a (i,j) = sin (a(i,j))
```

- Usar dispersión de índices, e.g.

```
forall (i=1,n, j=1:n) a (u(i),j) = a (u(j),i)
```

3.13.2. Directiva ejecutable `independent`

Esta directiva ejecutable se la debe intercalar en la parte ejecutable del programa, anteponiéndola a cada estamento `do` o `forall` que se intente paralelizar. Esta directiva ejecutable le asegura al compilador que no habría interferencia entre las iteraciones del lazo `do` o `forall`, directa o indirectamente. Por ejemplo,

- para los lazos `do`, la directiva ejecutable `independent` significa que las iteraciones y asignaciones pueden hacerse *en cualquier orden perdiendo la concepción secuencial*. Por ejemplo,

```
!hpf$ independent
do  i = 1, n
  a (i) = 4 * i + 3
end do
```

- en cambio, para los lazos `forall`, la directiva ejecutable `independent` significa que *no* es necesaria una sincronización al calcular toda el miembro derecho de la expresión (RHS) y el comienzo de asignación al miembro izquierdo (LHS), esto es, no todo el RHS tiene que ser evaluado *antes* de comenzar la asignación en el LHS tal como en

```
!hpf$ independent
forall (i = 1:n) a (i) = 4 * i + 3
```

- empero, ciertas operaciones tales como asignar más de una vez a un mismo elemento, estamentos `exit`, `stop`, `pause` e I/O externos, prescriben la independencia, por lo que el compilador lo convierte a un lazo secuencial.

3.13.3. Cómo saber si un lazo puede o no ser independiente

La posible dependencia de datos en los lazos conduce a responder la pregunta: *hay alguna iteración que dependa del resultado de la anterior?*, lo cual equivale a preguntarse: *se puede (o no) revertir el orden de las iteraciones sin que se altere el resultado?*. Esto último deriva en el siguiente criterio:

para saber si un lazo puede o no ser independiente: *probar con revertir el orden de ejecución del lazo y, si con el nuevo orden sigue dando el mismo resultado, entonces ese lazo es independiente.*

3.14. Funciones y procedimientos con atributo `pure`

Las funciones y procedimientos sin efectos colaterales indeseables pueden usarse dentro de un lazo paralelo, tal como un `forall` o un `do` precedido por la directiva `independent`. Todas las funciones intrínsecas y las que tienen el atributo `elemental` también son *puras*. Para que el compilador las acepte como tales hay que prefijar sus implementaciones con la palabra reservada `pure`, e.g. `pure function f (x,y)` o `pure subroutine g (x,y)`. Que sean *sin efectos colaterales* significa:

- ausencia de I/O externos (`read, write, print` ...) así como también `pause` o `stop`;
- ausencia de `allocate`;
- identificadas con el atributo `pure`;
- pueden ser llamados dentro de un `forall` y a su vez pueden llamar a otros procedimientos que sean puros;
- los argumentos mudos de las `function` deben tener el atributo `intent (in)` aunque se libera de esta restricción a las `subroutine`;
- los argumentos mudos no pueden ser alineados.

Como ejemplos consideremos el cálculo de la norma L_2, primero con una función pura

```
pure real function l2 (x,y)
  implicit none
  real, intent (in) :: x, y
  l2 = sqrt (x*x + y*y)
end function
...
forall (i=1,n, j=1:n) a (i,j) = l2 (x(i),x(j))
...
```

y ahora con un procedimiento puro

```
pure subroutine l2 (x,y,s)
  implicit none
  real, intent (in)   :: x, y
  real, intent (out) :: s
  s = sqrt (x*x + y*y)
end function
...
forall (i=1,n, j=1:n) call l2 (x (i), x (j), z (i,j) )
...
```

3.15. Matriz traspuesta usando transpose y forall

```
program  p13
  implicit none
  integer, parameter :: n = 4
  integer, dimension (n,n) :: a, b, c
  integer :: i, j, s=0
  !hpf$ distribute (block,block) :: a
  !hpf$ align b (j,i) with a (i,j) !no hay comunicacion
  !hpf$ align c (j,i) with a (i,j) !no hay comunicacion
  write (*,*) "Arreglos: traspuesta usando transpose y forall"
  do j = 1, n ; do i = 1, n
    s = s + 1 ; a (i,j) = s
  end do  ; end do
  b = transpose (a)
  forall (i=1:n,j=1:n) c (j,i) = a (i,j)
  write (*,*) "matriz A "
  do  i = 1, n ; write (*,100) " ", a (i,:) ; end do
  write (*,*) "matriz B "
  do  i = 1, n ; write (*,100) " ", b (i,:) ; end do
  write (*,*) "matriz C "
  do  i = 1, n ; write (*,100) " ", c (i,:) ; end do
  100 format (1x, a,10(1x,i3))
end program
```

3.16. Dispersión y matrices ralas en formatos CSC y CSR

Supongamos que se pide evaluar el producto matricial $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ dado por

$$\begin{bmatrix} 0 & a_{12} & 0 & 0 \\ a_{21} & 0 & a_{23} & 0 \\ a_{31} & 0 & a_{33} & 0 \\ 0 & a_{42} & 0 & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} ; \tag{3.1}$$

cuando la matriz $\boldsymbol{A}$ se guarda en algún formato *ralo*. Entre los propuestos en literatura[32, 18] consideremos los formatos ralos: completo, CSC y CSR (e.g. Blanco *et al.*[3].

3.16.1. Formato ralo completo

El formato ralo completo ("inocente") es, en general, poco eficiente, por lo que se usa pocas veces. De todas maneras, asumiendo tal formato, tendremos

$$
\begin{aligned}
i_a &= \left[\begin{array}{ccccccc} 1 & 2 & 2 & 3 & 3 & 4 & 4 \end{array} \right] ; \\
j_a &= \left[\begin{array}{ccccccc} 2 & 1 & 3 & 1 & 3 & 2 & 4 \end{array} \right] ; \\
a_a &= \left[\begin{array}{ccccccc} a_{12} & a_{21} & a_{23} & a_{31} & a_{33} & a_{42} & a_{44} \end{array} \right] ;
\end{aligned}
\tag{3.2}
$$

con los cuales es posible plantear las siguientes operaciones

$$
\begin{aligned}
x(j_a) &= \left[\begin{array}{ccccccc} x_2 & x_1 & x_3 & x_1 & x_3 & x_2 & x_4 \end{array} \right] ; \\
a_c x(j_a) &= \left[\begin{array}{ccccccc} a_{12}x_2 & a_{21}x_1 & a_{23}x_3 & a_{31}x_1 & a_{33}x_3 & a_{42}x_2 & a_{44}x_4 \end{array} \right] ;
\end{aligned}
\tag{3.3}
$$

Notar que en ambos arreglos i_a, j_a hay índices repetidos. Hasta aquí casi se tiene el resultado buscado para $\boldsymbol{y}$, solo resta sumar en los lugares apropiados para obtener

$$
y = \left[\begin{array}{cccc} (a_{12}x_2) & (a_{21}x_1 + a_{23}x_3) & (a_{31}x_1 + a_{33}x_3) & (a_{42}x_2 + a_{44}x_4) \end{array} \right] . \tag{3.4}
$$

Los lugares en y en donde deben acumularse los elementos del arreglo $a_a x(j_a)$ están dados justamente por el arreglo i_a. Para hacer esta operación en paralelo en HPF se dispone de la intrínseca `sum_scatter` para lo cual hay que incluir la sentencia `use hpf_library`. El programa `rala_csa` codifica este ejemplo.

```
program rala_csa
  use hpf_library
  implicit none
  integer, parameter :: n = 4, p = 7
  integer, dimension (n) :: x, y
  integer, dimension (p) :: a, b
  integer, dimension (p) :: i, j
  !hpf$ distribute (block)   :: a
  !hpf$ align (:) with a (:) :: b
  !hpf$ align (:) with a (:) :: i, j
  !hpf$ align (:) with a (:) :: x, y
  write (*,*) "Matrices ralas: formato completo (i,j,a_ij) "
  write (*,*) "producto y=Ax con scatter HPF"
  i = [ 1, 2, 2, 3, 3, 4, 4 ]
  j = [ 2, 1, 3, 1, 3, 2, 4 ]
  a = [ 5, 6, 7, 8, 9, 1, 2 ]
  x = [ 4, 5, 6, 7 ]
  b = a * x (j)
  y (1:n) = sum_scatter (b (1:p), y (1:n), i (1:p) )
end program
```

3.16.2. Formato *Compressed Sparse Column* (CSC)

El formato ralo **CSC** del ejemplo es

$$
\begin{aligned}
j_c &= \left[\begin{array}{ccccc} 1 & 3 & 5 & 7 & 8 \end{array} \right] ; \\
i_c &= \left[\begin{array}{ccccccc} 2 & 3 & 1 & 4 & 2 & 3 & 4 \end{array} \right] ; \\
a_c &= \left[\begin{array}{ccccccc} a_{21} & a_{32} & a_{12} & a_{41} & a_{23} & a_{33} & a_{44} \end{array} \right] ;
\end{aligned}
\tag{3.5}
$$

y planteamos las siguientes operaciones:

$$\begin{aligned}
b(j_c) = x \;\rightarrow\; b &= \left[\begin{array}{ccccccc} x_1 & 0 & x_2 & 0 & x_3 & 0 & x_4 \end{array}\right] ; \\
s &= \left[\begin{array}{ccccccc} T & T & F & F & T & T & F \end{array}\right] ; \\
c = \text{copy_prefix (b,s)} &= \left[\begin{array}{ccccccc} x_1 & x_1 & x_2 & x_2 & x_3 & x_3 & x_4 \end{array}\right] ; \\
d = a_c c &= \left[\begin{array}{ccccccc} a_{21}x_1 & a_{31}x_1 & a_{12}x_2 & a_{42}x_2 & a_{23}x_3 & a_{33}x_3 & a_{44}x_4 \end{array}\right] ; \\
y = \text{sum_scatter}(d, y, i_c) &= \left[\begin{array}{cccc} (a_{12}x_2) & (a_{21}x_1 + a_{23}x_3) & (a_{31}x_1 + a_{33}x_3) & (a_{42}x_2 + a_{44}x_4) \end{array}\right] .
\end{aligned} \tag{3.6}$$

En este caso introducimos el arreglo booleano `s` para identificar cada *columna* de la matriz `A`, esto es, mientras no se produzca un salto en el valor booleano, estaremos en la misma columna. Para obtener el arreglo `c` en paralelo en HPF se dispone de la instrínseca `copy_prefix`. (ver demo `rala_csc`).

```
program rala_csc
  use hpf_library
  implicit none
  integer, parameter :: n = 4, p = 7
  integer, dimension (n)    :: x, y
  integer, dimension (n+1) :: j
  integer, dimension (p)    :: i
  integer, dimension (p)    :: a, b, c, d
  logical, dimension (p)    :: s
  !hpf$ distribute (block)    :: a
  !hpf$ align (:) with a (:) :: b, c, d
  !hpf$ align (:) with a (:) :: i, s
  !hpf$ align (:) with a (:) :: x, y
  integer :: k, l
  logical :: tira
  write (*,*) "Matrices ralas: formato CSC (Compressed Sparse Column)"
  write (*,*) "producto y=Ax con scatter HPF"
  !caso n = 4, p = 7
  j = [ 1, 3, 5, 7, 8 ]
  i = [ 2, 3, 1, 4, 2, 3, 4 ]
  a = [ 6, 8, 5, 1, 7, 9, 2 ]
  x = [ 4, 5, 6, 7 ]
  !construye segmentos booleanos para identificar cada columna
  tira = .true.
  do  k = 1, n
   do  l = j (k), j (k + 1) - 1
     s (l) = tira
   end do
   tira = .not. tira
  end do
  b (j (1:n)) = x
  c = copy_prefix (b, segment = s)
  d = a * c ; y = 0
  y (1:n) = sum_scatter (d (1:p), y (1:n), i (1:p) )
end program
```

3.16.3. Formato CSR (*Compressed Sparse Row*)

El formato ralo **CSR** del ejemplo es

$$\begin{aligned} i_r &= \left[\begin{array}{ccccc} 1 & 2 & 4 & 6 & 8 \end{array}\right] ; \\ j_r &= \left[\begin{array}{ccccccc} 2 & 1 & 3 & 1 & 3 & 2 & 4 \end{array}\right] ; \\ a_r &= \left[\begin{array}{ccccccc} a_{12} & a_{21} & a_{23} & a_{31} & a_{33} & a_{42} & a_{44} \end{array}\right] ; \end{aligned} \tag{3.7}$$

y planteamos las siguientes operaciones:

$$\begin{aligned} b = x(j_r) \quad \rightarrow \quad b &= \left[\begin{array}{ccccccc} x_2 & x_1 & x_3 & x_1 & x_3 & x_2 & x_4 \end{array}\right] ; \\ s &= \left[\begin{array}{ccccccc} T & F & F & T & T & F & F \end{array}\right] ; \\ c = a_r b &= \left[\begin{array}{ccccccc} a_{12}x_2 & a_{21}x_1 & a_{23}x_3 & a_{31}x_1 & a_{33}x_3 & a_{42}x_2 & a_{44}x_4 \end{array}\right] ; \\ d = \text{sum_suffix}(c, s) &= \left[\ (a_{12}x_2) \quad (a_{21}x_1 + a_{23}x_3) \quad (a_{23}x_3) \quad (a_{31}x_1 + a_{33}x_3) \right. \\ & \qquad \left. (a_{33}x_3) \quad (a_{42}x_2 + a_{44}x_4) \quad (a_{44}x_4)\ \right] . \end{aligned} \tag{3.8}$$

Ahora el arreglo booleano `s` identifica cada *fila* de la matriz `A`, esto es, mientras no se produzca un salto en el valor booleano, estamos en la misma fila. Para obtener el arreglo `d` en paralelo en HPF se dispone de la instrínseca `sum_suffix`, asistida en este caso por el segmento booleano `s`. El programa demo `rala_csr` codifica este ejemplo.

```
program rala_csr
  use hpf_library
  implicit none
  integer, parameter :: n = 4, p = 7
  integer, dimension (n)   :: x, y
  integer, dimension (n+1) :: i
  integer, dimension (p)   :: j
  integer, dimension (p)   :: a, b, c, d
  logical, dimension (p)   :: s
  !hpf$ distribute (block)   :: a
  !hpf$ align (:) with a (:) :: b, c, d
  !hpf$ align (:) with a (:) :: i, s
  !hpf$ align (:) with a (:) :: x, y
  integer :: k, l
  logical :: tira
  write (*,*) "Matrices ralas: formato CSR (Compressed Sparse Row)"
  write (*,*) "producto y=Ax con scatter HPF"
  i = [ 1, 2, 4, 6, 8 ]
  j = [ 2, 1, 3, 1, 3, 2, 4 ]
  a = [ 5, 6, 7, 8, 9, 1, 2 ]
  x = [ 4, 5, 6, 7 ]
  ! construye segmentos booleanos para identificar cada fila
  tira = .true.
  do  k = 1, n
   do  l = i (k), i (k + 1) - 1
     s (l) = tira
   end do
   tira = .not. tira
  end do
  b = x (j (1:p))
  c = a * b
  d (1:p) = sum_suffix (c (1:p), segment = s)
  y = d (i(1:n))
end program
```

3.16.4. Ausencia de dispersión con índices repetidos en F95

Nota: **NO** se disponen de procedimientos de dispersión en la librería del F95 cuando hay índices repetidos, por lo que hay que implementarlos en forma escalar. Por ejemplo, se puede emular el `sum_scatter` con la siguiente subrutina:

```
subroutine scatter_add_d (b, a, ib)
  real     (idp), dimension (:), intent (out) :: b
  real     (idp), dimension (:), intent (in)  :: a
  integer (iin), dimension (:), intent (in)  :: ib
  integer (iin) :: m, n, j, i
  n = siza (size (a), size (ib), "scatter_add") !funcion de usuario
  m = size (b) ; b = 0.0d0
  do  j = 1, n
    i = ib (j)                        !indice destino
    if (i < 1 .or. i > m) cycle  !esta fuera de rango
    b (i) = b (i) + a (j)
  end do
end subroutine
```

3.17. Compilación con ADAPTOR

Usaremos el traductor ADAPTOR para compilar los programas F95-HPF. Los programas usualmente los particionaremos usando módulos (`module`) guardados en disco como archivos independientes e identificados por el prefijo `m`, en este caso como una convención personal del usuario, que serán usados por un programa principal `xx.f90`. Usaremos alocamiento de memoria *dinámica*. Notar que el `module` es el símil de las clases disponible desde F90 pero, hasta F95 inclusive, son algo restrictivas como para hacer POO, pues son estáticas y no permiten herencia dinámica nativa (sin emulación). Por ejemplo, para resolver un Sistema de Ecuaciones Lineales (SEL) $\boldsymbol{Ax} = \boldsymbol{b}$ usando la factorización LU, en donde asumiremos que la matriz del sistema $\boldsymbol{A}$ es regular, podemos proceder de la siguiente manera:

- La compilación de cada módulo por separado, respetando la jerarquía estática de los módulos, la haremos con

```
adaptor -hpf -dm -free -c m_ctes.f90
adaptor -hpf -dm -free -c m_temps0.f90
adaptor -hpf -dm -free -c m_tools.f90
adaptor -hpf -dm -free -c m_gauss1.f90
adaptor -hpf -dm -free -c gauss.f90
adaptor -hpf -dm -free -o gauss.exe *.o
```

- Luego de obtener el binario, se lo ejecuta en el *cluster Beowulf* mediante comandos de `MPICH` tales como

```
mpdboot -n 21 -f ~/mpd.hosts
mpdtrace
mpiexec -machinefile ~/machi.dat -1 -np 20 gauss1.exe
mpdallexit
```

3.18. Módulos para las constantes y herramientas

El módulo de las constantes globales `m_ctes` contiene aquellas que no se pueden modificar durante la ejecución del programa, y que son definidas en la fase de compilación. Al menos incluye lo siguiente

```
module m_ctes
  implicit none
  public
  !parametros de precision usuales
  integer, parameter :: iin = kind (1)
  integer, parameter :: isp = kind (1.0e0)
  integer, parameter :: idp=kind(1.0d0)
  integer, private, parameter :: q=selected_real_kind(2*precision(1.0_idp))
  integer, parameter :: iqp = (1+sign(1,q))/2*q+(1-sign(1,q))/2*idp
end module
```

Por otra parte, el módulo `m_temps0` incluye auxiliares para verificar el alocamiento de memoria dinámica

```
module m_temps
  use  m_ctes
  implicit none
  private          !por omision todo es asi
  public nerror, ier
  !para verificar alocamiento de memoria RAM
  integer (iin), parameter :: nerror = 64
  integer (iin), dimension (nerror) :: ier
end module
```

mientras que en el módulo `m_tools0` se declaran y definen herramientas varias tales como el `siza`, para validar coherencia de las dimensiones de los arreglos, `outer_product` con el producto exterior de dos vectores, `scatter_add` con la suma con dispersión y el `swap` para intercambiar arreglos:

```
module m_tools
 use   m_ctes
 use   m_temps
 implicit none
 interface siza
   module procedure siza2, siza3
 end interface
 interface outer_prod
   module procedure outer_prod_z, outer_prod_d
 end interface
 interface scatter_add
   module procedure scatter_add_z, scatter_add_d
 end interface
 interface swap
   module procedure zswap, dswap
 end interface
 public  !todo es asi excepto lo indicado en contrario
 character(10), private, parameter :: f = 'm_tools:> '
 contains
 !informa y fin si los enteros no son todos iguales
 function  siza2 (n1, n2, s)
  integer (iin), intent (in) :: n1, n2
  character(*) , intent (in) :: s
  integer (iin) :: siza2
  if (n1 .eq. n2) then
   siza2 = n1
  else
   write (*,*) "n1 n2 ", n1, n2
   write (*,100) " error (siza2): fallo en " // s
   stop
  end if
  if (n1 < 0)  stop" error (siza2): n < 0 "
```

```
 100 format (a)
end function
function  siza3 (n1, n2, n3, s)
 integer (iin), intent (in) :: n1, n2, n3
 character(*) , intent (in) :: s
 integer (iin) :: siza3
 if (n1 .eq. n2 .and. n2 .eq. n3) then
  siza3 = n1
 else
  write (*,*)" n1 n2 n3 ", n1, n2, n3
  write (*,100) " error (siza3): fallo en " // s
  stop
 end if
 if (n1 < 0)  stop" error (siza3): n < 0 "
 100 format (a)
end function
!producto exterior
function  outer_prod_z (za,zb)
  complex (idp), dimension (:), intent (in) :: za, zb
  complex (idp), dimension (size(za),size(zb)) :: outer_prod_z
  outer_prod_z = spread (za, dim = 2, ncopies = size (zb) ) * &
                 spread (zb, dim = 1, ncopies = size (za) )
end function
function outer_prod_d (da,db)
  real (idp), dimension (:), intent (in) :: da, db
  real (idp), dimension (size(da),size(db)) :: outer_prod_d
  outer_prod_d = spread (da, dim = 2, ncopies = size (db) ) * &
                 spread (db, dim = 1, ncopies = size (da) )
end function
!suma expandida: desde "a" hacia "b" en las posiciones destino "ib"
!en HPF se la puede reemplazar por la "sum_scatter"
subroutine  scatter_add_z (zb, za, bindex)
 complex (idp), dimension (:), intent (out) :: zb
 complex (idp), dimension (:), intent (in)  :: za
 integer (iin), dimension (:), intent (in)  :: bindex
 integer (iin) :: m, n, j, i
 n = siza (size (za), size (bindex), "scatter_add")
 m = size (zb)
 zb = cmplx (0.0,0.0)
 do  j = 1, n
   i = bindex (j)               !indice destino
   if (i < 1 .or. i > m) cycle  !esta fuera de rango
   zb (i) = zb (i) + za (j)
 end do
end subroutine
subroutine  scatter_add_d (db, da, bindex)
 real    (idp), dimension (:), intent (out) :: db
 real    (idp), dimension (:), intent (in)  :: da
 integer (iin), dimension (:), intent (in)  :: bindex
 integer (iin) :: m, n, j, i
 n = siza (size (da), size (bindex), "scatter_add")
 m = size (db)
 db = 0.0
 do  j = 1, n
   i = bindex (j)               !indice destino
   if (i < 1 .or. i > m) cycle  !esta fuera de rango
   db (i) = db (i) + da (j)
 end do
end subroutine
!swap
subroutine  zswap (za,zb)
  complex (idp), dimension (:), intent (inout) :: za, zb
  complex (idp), dimension (size(za)) :: zt
  zt = za
  za = zb
  zb = zt
end subroutine
subroutine  dswap (da,db)
  real (idp), dimension (:), intent (inout) :: da, db
```

```
    real (idp), dimension (size(da)) :: dt
    dt = da
    da = db
    db = dt
  end subroutine
end module
```

3.19. Factorización LU

El módulo `m_gauss1` contiene la factorización LU con pivotaje parcial, substitución hacia adelante y hacia atrás, donde para la matriz distribuida se asume un mapeo HPF de tipo descriptivo:

```
module m_gauss1
  use   m_ctes
  use   m_temps
  use   m_tools1
  implicit none
  private               !por omision todo es privado
  !precision double    !with descriptive Mapping
  public  dsolve_b1_adp
  public  dlu_factor_adp
  public  dforward_sol_adp
  public  dbackward_sol_adp
  !para medir tiempos
  real    (idp), dimension (3) :: time
  integer (iin) :: h1, h2
  contains
    subroutine  dsolve_b1_adp (a, b, imprime)
    (...)
    end subroutine
    subroutine  dlu_factor_adp (a, d, indx)
    (...)
    end subroutine
    subroutine  dforward_sol_adp (a, b, y, indx)
    (...)
    end subroutine
    subroutine  dbackward_sol_adp (a, y, x)
    (...)
    end subroutine
    subroutine  mflops_g (time, n, m, raiz)
    (...)
    end subroutine
```

Luego, la subrutina que conduce al *solver* es de la forma

```
  subroutine  dsolve_b1_adp (a, b, imprime)
    integer (iin) :: i, j, k, p, eco
    real    (idp), dimension (:,:), intent (inout) :: a
    real    (idp), dimension (:)  , intent (inout) :: b
    logical      , optional       , intent (in)    :: imprime
    real    (idp), dimension (:)  , allocatable :: y
    real    (idp), dimension (:)  , allocatable :: d
    integer (iin), dimension (:)  , allocatable :: indx
    integer (iin) :: n, m
    !hpf$ distribute * (*,block) :: a
    !hpf$ align (i)   with * a (i,*) :: b
    !hpf$ align (i)   with   a (i,*) :: y
    !hpf$ align (i)   with   a (i,*) :: d, indx
    !valida tamanios
    n = siza (size (b), size (a,1), size (a,2), "size (a)" )
    !aloca
    ier = 0
    allocate (y    (1:n), stat = ier (1) )
    allocate (d    (1:n), stat = ier (2) )
    allocate (indx (1:n), stat = ier (3) )
    if (any (ier .ne. 0)) call errata (ier(1:3), "solve: aloca")
    if  ( present (imprime) ) then
```

```
    eco = 0 ; if (imprime) eco = 1
  end if
  !tareas
  call     dlu_factor_adp (a, d, indx)
  call   dforward_sol_adp (a, b, y, indx)
  call  dbackward_sol_adp (a, y, b)
  !estadistica
  if  (eco == 1) then
    m = 1 ; call  mflops_g (time, n, m, "gauss_pivo")
  end if
  !dloca
  ier = 0
  deallocate (indx, stat = ier (3) )
  deallocate (d   , stat = ier (2) )
  deallocate (y   , stat = ier (1) )
  if (any(ier.ne.0)) call errata (ier(1:3), "solve: dloca")
end subroutine
```

Por su parte, la factorización LU con pivotaje está dada por

```
subroutine  dlu_factor_adp (a, d, indx)
  integer (iin) :: i, j, k, n,kmax
  real    (idp), dimension (:,:), intent (inout) :: a
  real    (idp), dimension (:)  , intent (out)   :: d
  integer (iin), dimension (:)  , intent (out)   :: indx
  real    (idp), parameter :: tenue = 1.0d-20
  real    (idp), dimension (size(a,1)) :: t
  !hpf$ distribute * (*,block) :: a
  !hpf$ align (i)   with * a (i,*) :: d, indx
  !hpf$ align (i)   with   a (i,*) :: t
  !valida tamanios
  n = siza (size (a,1), size (a,2), "size (a)" )
  call system_clock (count = h1)
  !scaling
  indx = 0
  d = maxval (abs (a), dim = 2) !loop on rows
  if (any (d == 0.0_idp) ) stop" singular matrix ... "
  d = 1.0_idp / d
  do  k = 1, n
    !partial pivoting
    kmax = (k - 1) + maxlocat (d (k:n) * abs ( a (k:n,k) ) )
    if (kmax .ne. k) then
      t (1:n)      = a (kmax,1:n)
      a (kmax,1:n) = a (k,1:n)
      a (k,1:n)    = t (1:n)
      d (kmax) = d (k)
    end if
    indx (k) = kmax
    if  (a (k,k) == 0.0_idp)  a (k,k) = tenue
    !reduction with outer product
    a (k+1:n,k:k)   = a (k+1:n,k:k) / a (k,k)
    a (k+1:n,k+1:n) = a (k+1:n,k+1:n) -                         &
            spread (a (k+1:n,k), dim = 2, ncopies = n-k) *   &
            spread (a (k,k+1:n), dim = 1, ncopies = n-k)
  end do
  call system_clock (count = h2)
  time (1) = dble (h2 - h1)
end subroutine
```

Luego de factorizar, hacemos la solución hacia adelante con

```
subroutine  dforward_sol_adp (a, b, y, indx)
  integer (iin) :: i, j, k, l, n
  real    (idp), dimension (:,:), intent (in)    :: a
  real    (idp), dimension (:)  , intent (inout) :: b
  real    (idp), dimension (:)  , intent (out)   :: y
  integer (iin), dimension (:)  , intent (inout) :: indx
  real    (idp) :: prima, s
  character (22) :: e = "size (a) | length (b,y)"
  !hpf$ distribute * (*,block) :: a
  !hpf$ align (i)   with * a (i,*) :: b, y, indx
```

```
    !valida tamanios
    n = siza (size (b), size (y), size (a,1), size (a,2), e)
    if (any (indx < 1)) stop "error (forward_sol): index  < 1 "
    if (any (indx > n)) stop "error (forward_sol): index  > n "
    call system_clock (count = h1)
    k = 0
    do  i = 1, n
     l      = indx (i)
     prima = b (l)
     b (l) = b (i)
     s = 0.0_idp
     if      (k .ne. 0) then
       s = sum ( a (i,k:i-1) * y (k:i-1) )
     elseif (prima .ne. 0.0_idp) then
       k = i
     end if
     y (i) = prima - s
    end do
    call system_clock (count = h2)
    time (2) = dble (h2 - h1)
  end subroutine
```

Y, a continuación, la solución hacia atrás

```
  subroutine  dbackward_sol_adp (a, y, x)
    integer (iin) :: i, j, k, n
    real     (idp), dimension (:,:), intent (in)   :: a
    real     (idp), dimension (:)  , intent (in)   :: y
    real     (idp), dimension (:)  , intent (out) :: x
    real     (idp) :: s
    character (22) :: e = "size (a) | length (x,y)"
    !hpf$ distribute * (*,block) :: a
    !hpf$ align (i)   with * a (i,*) :: y, x
    !alida tamanios
    n = siza (size (x), size (y), size (a,1), size (a,2),e)
    call system_clock (count = h1)
    do  i = n, 1, -1
      s = sum ( a (i,i+1:n) * x (i+1:n) )
      x (i) = (y (i) - s) / a (i,i)
    end do
    call system_clock (count = h2)
    time (3) = dble (h2 - h1)
  end subroutine
```

La última subrutina hace la estadística

```
  subroutine  mflops_g (time, n, m, raiz)
    real     (idp), dimension (:), intent (inout) :: time
    integer (iin), intent (in) :: n      !orden de la matriz
    integer (iin), intent (in) :: m      !nro de cargas
    character (*), intent (in) :: raiz  !raiz archivo
    character (len(raiz)+4) :: arch     !extension prefijada = 4
    character (4) :: exte = ".tim"      !aqui la exten. es cte
    character (44):: s1, s2
    integer (iin) :: l, p, ntime, h3
    real     (idp) :: total, ops, mflops
    !nro de procesadores
    p = number_of_processors ()          !solo en HPF
    !control
    ntime = size (time)
    if (ntime < 3) stop "error (mflops_g): size (time) < 3 "
    !pasa a segundos
    call system_clock (count_rate = h3)  !cte para pasar a seg
    if (h3 < 1) h3 = 1                   !control
    time = time / dble (h3)              !pasa de ciclos a seg
    !tiempo total
    total = sum (time)
    !estadistica
    ops = (2.0/3.0) * dble (n) ** 3 + 2.0 * dble (n) ** 2
    mflops = ops / (1.0e6 * total)
```

```
    !define archivo
    l = len_trim (raiz)        !longitud omitiendo blancos
    arch = raiz (1:l) // exte  !apendiza en ese orden sin blancos
    !resumen a disco
    s1 ="  p        n   m   twall_factorLU   twall_"
    s2 ="forward  twall_back          Mflops     "
    print 100, " archivo timer: " // arch
    print 100, s1 // s2
    open (1, file = arch          , &
             status = "unknown"   , &
             position = "append")
    write (*,110) p, n, m, time (1:3), mflops
    write (1,110) p, n, m, time (1:3), mflops
    close (1, status = 'keep')
    print 120, total, mflops
    100 format (a)
    110 format (1x, i2, 1x, i8, 1x, i3, 3(1x,e16.8), 1x, f12.3 )
    120 format (" elapsed time = ",e16.8," seconds ;",f12.3," Mflops")
  end subroutine
end module
```

Finalmente tendremos el siguiente programa principal

```
program  dgauss1v
  use m_ctes
  use m_temps
  use m_tools1
  use m_gauss1
  implicit none
  integer (iin) :: i, j, k, n
  real    (idp), dimension (:,:), allocatable ::  a
  real    (idp), dimension (:)  , allocatable ::  b, r
  real    (idp), dimension (:,:), allocatable :: a0
  real    (idp), dimension (:)  , allocatable :: b0
  real    (idp), dimension (:)  , allocatable ::  x
  real    (idp) :: s
  !hpf$ distribute (*,block) :: a
  !hpf$ align (i,*) with a (i,*) :: a0
  !hpf$ align (i)   with a (i,*) :: b0, b,  r
  integer (iin), parameter :: m = 1
  integer (iin) :: n_unknowns, n_loads
  integer (iin) :: falla = 1
  namelist /gauss_basic_data/ n_unknowns, n_loads
  !lee del disco
  open   (1, file = "gauss_basic.dat", status = "old", err = 100)
  read   (1,nml = gauss_basic_data)
  write  (*,nml = gauss_basic_data)
  close  (1, status = "keep")
  n = n_unknowns
  !carteles
  write (*,*)
  write (*,*) "Gauss solution of a SEL with dynamic RAM "
  write (*,*) "number of unknows ; n = ", n
  write (*,*) "number of loads   ; m = ", m
  !aloca sistema
  ier = 0
  allocate ( a (1:n,1:n), stat = ier (1) )
  allocate ( b (1:n)    , stat = ier (2) )
  allocate (a0 (1:n,1:n), stat = ier (3) )
  allocate (b0 (1:n)    , stat = ier (4) )
  allocate ( r (1:n)    , stat = ier (5) )
  allocate ( x (1:n)    , stat = ier (6) )
  if ( any (ier .ne. 0) ) call errata (ier(1:6), "aloca")
  !define un SEAL A x = b
  do  j = 1, n
    call  random_number (x) ; a (1:n,j) = x (1:n)
  end do
  call  random_number (x) ; b (1:n) = x (1:n)
  !hpf$ independent
```

```
  forall (i=1:n)  a (i,i) = a (i,i) + dble (n)
  !copia para posterior verificacion de los resultados
  a0 = a  ;  b0 = b
  !solucion gaussiana
  call  dsolve_b1_adp (a, b, imprime = .true.)
  !verifica matriz solucion X con R = B - A X
  r = matmul (a0,b) - b0 ; s = sqrt (sum (abs (r * r)))
  print *, "|| r ||_2 = ", s
  !dloca
  ier = 0
  deallocate ( x, stat = ier (6) )
  deallocate ( r, stat = ier (5) )
  deallocate (b0, stat = ier (4) )
  deallocate (a0, stat = ier (3) )
  deallocate ( b, stat = ier (2) )
  deallocate ( a, stat = ier (1) )
  if (any (ier .ne. 0))  call errata (ier(1:7), "dloca")
  falla = 0
  100 if (falla .ne. 0) stop "error: namelist file was not found ..."
  110 format (a)
  120 format (1x, i4, 1x, i20, 1x, e24.12)
end program
```

3.20. Conjunto de Mandelbrot

El conjunto de Mandelbrot es el conjunto de números complejos que se obtiene luego de iterar un cierto número de veces la función $z^2 + c$, donde $z = z' + iz''$ y $c = c' + ic''$ es una constante compleja tal que $-1 \leq c', c'' \leq 1$, aunque en el demo se restringe al primer cuadrante, ver Marshall[26], pág. 199. Es decir, iteramos con $z_{k+1} = z_k^2 + c$, con condición inicial $z_0 = 0$, para $k = 0, 1, ...k_{\text{máx}}$, hasta que $|z_k| > s_{\text{máx}}$, o bien $k > k_{\text{máx}}$, con s_{max} dado y $k_{\text{máx}} = 255$, o algún otro número máximo conveniente. Los números entre 0 y $k_{\text{máx}}$ representan colores. En caso de programar usando sólo arreglos reales hay que separar en sus partes real $z'_{k+1} = (z_k'^2 - z_k''^2) + c'$ e imaginaria $z''_{k+1} = 2z'_k z''_k + c''$. Usualmente se grafica el número de iteraciones necesario k para que $|z_k|$ alcance cierta cota (e.g. iterar mientras $|z_k| < 4$), con $k = 0, 1, ..., k_{max}$. Luego, se lo convierte a escala de colores y se lo grafica en función de c', c''. Para esta tarea usamremos simplemente una distribución en bloques.

```
program mandel
  implicit none
  integer, parameter :: n = 2048, resol = 516
  integer, dimension (n,n) :: color
  real    , dimension (n,n) :: zr, zi, cr, ci, ur, ui
  !hpf$ distribute (block,block) :: zr, zi, cr, ci, ur, ui, color
  real     :: s10, s11, s15, s20, s25, s30, s40, s50, s60
  integer :: i, j, k, c, rojo, verde, azul
  !inicializa y calcula
  forall (i=1:n, j=1:n)
    zr (i,j) = real (i-1) / real (n-1)
    zi (i,j) = real (j-1) / real (n-1)
  end forall
  cr = zr ; ci = zi ; ur = zr * zr ; ui = zi * zi ; color = 0
  resolu : do   k = 0, resol
    where ( (ur + ui) .le. 4.0 )
      ur = zr * zr              ;  ui = zi * zi
      zi = 2.0 * zr * zi + ci   ;  zr = (ur - ui) + cr
      color = k
    end where
  end do
  !archivo para gv en monocolor
  open  (10, file = 'mandel_bw.pgm', status = 'unknown')
  write (10,100)  n, n, resol
  write (10,110)  color
  close (10, status = 'keep')
  !archivo para gv en color
  s10 = 0.04 * resol ; s11 = 0.04 * resol
  s15 = 0.06 * resol ; s20 = 0.08 * resol
  s25 = 0.10 * resol ; s30 = 0.12 * resol
  s40 = 0.16 * resol ; s50 = 0.20 * resol ; s60 = 0.24 * resol
  open  (11, file = 'mandel_co.pgm', status = 'unknown')
  write (11,120)  n, n, resol
  do  i = 1, n ; do  j = 1, n
    c = color (i,j)
    if     ( c .eq. resol ) then
      rojo = 0          ; verde = 0       ; azul = 0
    elseif ( c .ge.   0 .and. c .le. s10 ) then
      rojo = c * s25    ; verde = c * s20 ; azul = c * s10
    elseif ( c .ge. s11 .and. c .le. s15 ) then
      rojo = c * s60    ; verde = c * s15 ; azul = c * s30
    else
      rojo = resol / 1 ; azul  = resol / 2 ; verde = resol / 4
    end if
    write (11,130) rojo, verde, azul
  end do ; end do
  close (11, status = 'keep')
  !formatos
  100 format ('P2' / i5, 2x, i5 / i5)
  110 format ( 11 (1x, i5) )
  120 format ('P3' / i5, 2x, i5 / i5)
  130 format ( 11 (1x, i5) )
end program
```

3.21. Apéndice: Arnoldi basado en Gram-Schmidt modificado

3.21.1. Deducción del algoritmo básico

Sean un $\boldsymbol{v}_1 \in \mathrm{R}^{\mathrm{n},1}$ tal que $||v_1||_2 = 1$ y la matriz regular $\boldsymbol{A} \in \mathrm{R}^{\mathrm{n},\mathrm{n}}$. Si empezamos haciendo $\boldsymbol{c}_2 = \boldsymbol{A}\boldsymbol{v}_1$ tendremos, en general, que $\boldsymbol{c}_2$ no es ortonormal a $\boldsymbol{v}_1$, es decir, el producto interno $(\boldsymbol{c}_2, \boldsymbol{v}_1)$ no es nulo ni $||\boldsymbol{c}_2||_2 \neq 1$. Entonces recurrimos a la ortonormalización de Gram-Schmidt haciendo $\tilde{\boldsymbol{c}}_2 = \boldsymbol{c}_2 - (\boldsymbol{c}_2, \boldsymbol{v}_1)\boldsymbol{v}_1$ y luego $\boldsymbol{v}_2 = \tilde{\boldsymbol{c}}_2/\beta_2$, donde $\beta_2 = ||\tilde{\boldsymbol{c}}_2||_2$, resultando $(\boldsymbol{v}_i, \boldsymbol{v}_j) = \delta_{ij}$, con $1 \leq i, j \leq 2$. Es inmediato verificar que ahora $\boldsymbol{v}_2$ es ortogonal a $\boldsymbol{v}_1$ pues hacemos $(\tilde{\boldsymbol{c}}_2, \boldsymbol{v}_1) = (\boldsymbol{c}_2, \boldsymbol{v}_1) - (\boldsymbol{c}_2, \boldsymbol{v}_1)(\boldsymbol{v}_1, \boldsymbol{v}_1) \equiv 0$.

Si repetimos el proceso tendremos $\boldsymbol{c}_3 = \boldsymbol{A}\boldsymbol{v}_2$ pero ahora $\boldsymbol{c}_3$ no es ortonormal a la base $\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$, es decir, en general $(\boldsymbol{c}_3, \boldsymbol{v}_1) \neq 0$, $(\boldsymbol{c}_3, \boldsymbol{v}_2) \neq 0$ ni $||\boldsymbol{c}_3||_2 \neq 1$. Entonces usamos de nuevo a Gram-Schmidt haciendo $\tilde{\boldsymbol{c}}_3 = \boldsymbol{c}_3 - (\boldsymbol{c}_3, \boldsymbol{v}_1)\boldsymbol{v}_1 - (\boldsymbol{c}_3, \boldsymbol{v}_2)\boldsymbol{v}_2$ y luego $\boldsymbol{v}_3 = \tilde{\boldsymbol{c}}_3/\beta_3$, donde $\beta_3 = ||\tilde{\boldsymbol{c}}_3||_2$, resultando $(\boldsymbol{v}_i, \boldsymbol{v}_j) = \delta_{ij}$, con $1 \leq i, j \leq 3$. Es inmediato verificar que ahora $\boldsymbol{v}_3$ es ortogonal a la base $\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$ pues hacemos $(\tilde{\boldsymbol{c}}_3, \boldsymbol{v}_1) = (\boldsymbol{c}_3, \boldsymbol{v}_1) - (\boldsymbol{c}_3, \boldsymbol{v}_1)(\boldsymbol{v}_1, \boldsymbol{v}_1) - (\boldsymbol{c}_3, \boldsymbol{v}_2)(\boldsymbol{v}_2, \boldsymbol{v}_1) \equiv 0$ y $(\tilde{\boldsymbol{c}}_3, \boldsymbol{v}_2) = (\boldsymbol{c}_3, \boldsymbol{v}_2) - (\boldsymbol{c}_3, \boldsymbol{v}_1)(\boldsymbol{v}_1, \boldsymbol{v}_2) - (\boldsymbol{c}_3, \boldsymbol{v}_2)(\boldsymbol{v}_2, \boldsymbol{v}_2) \equiv 0$. En general, para un $m \leq n$ dado y, usualmente, con $m \ll n$, haremos para $j = 1, 2, ..., m$

$$
\begin{aligned}
&\tilde{\boldsymbol{c}}_{j+1} = \boldsymbol{A}\boldsymbol{v}_j - \sum_{i=1}^{j} (\boldsymbol{v}_i, \boldsymbol{A}\boldsymbol{v}_j) \ ; \\
&\beta_{j+1} = ||\boldsymbol{c}_{j+1}||_2 \ ; \\
&\boldsymbol{v}_{j+1} = \tilde{\boldsymbol{c}}_j / \beta_{j+1} \ ;
\end{aligned}
\tag{3.9}
$$

el cual puede re-escribirse como

$$
\begin{aligned}
&\text{for } j = 1, 2, ..., m \text{ do} \\
&\quad \boldsymbol{t} = \boldsymbol{A}\boldsymbol{v}_j \\
&\quad \text{for } i = 1, 2, ..., j \text{ do } \tilde{h}_{ij} = (\boldsymbol{v}_i, \boldsymbol{t}) \\
&\quad \boldsymbol{s} = \sum_{i=1}^{j} \tilde{h}_{ij} \boldsymbol{v}_i \\
&\quad \boldsymbol{c}_j = \boldsymbol{t} - \boldsymbol{s} \\
&\quad \tilde{h}_{j+1,j} = ||\boldsymbol{c}_j||_2 \\
&\quad \text{if } (\tilde{h}_{j+1,j} < \varepsilon) \text{ exit} \\
&\quad \boldsymbol{v}_{j+1} = \boldsymbol{c}_j \ / \ \tilde{h}_{j+1,j}
\end{aligned}
\tag{3.10}
$$

dando lugar al algoritmo de Arnoldi [2]. Por ejemplo, cuando $m = 3$ tendremos $\tilde{\boldsymbol{H}}_3 \in \mathrm{R}^{3+1,3}$, $\boldsymbol{V}_{3+1} \in \mathrm{R}^{\mathrm{n},3+1}$, con

$$
\tilde{\boldsymbol{H}}_3 = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ \beta_1 & h_{22} & h_{23} \\ 0 & \beta_2 & h_{33} \\ 0 & 0 & \beta_3 \end{bmatrix} \quad ; \quad \boldsymbol{V}_4 = \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 & \boldsymbol{v}_3 & \boldsymbol{v}_4 \end{bmatrix} \quad ; \quad (\boldsymbol{v}_i, \boldsymbol{v}_j) = \delta_{ij} \quad 1 \leq i, j \leq 4 \ ; \tag{3.11}
$$

3.21.2. Algunas propiedades

Por ejemplo, cuando $m = 3$ tendremos de la Ec. 3.11

$$\boldsymbol{V}_3 = \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 & \boldsymbol{v}_3 \end{bmatrix} \qquad ; \qquad \boldsymbol{V}_3^T = \begin{bmatrix} \boldsymbol{v}_1^T \\ \boldsymbol{v}_2^T \\ \boldsymbol{v}_3^T \end{bmatrix} \ . \tag{3.12}$$

Como $\boldsymbol{V}_3$ es una base ortonormal resulta que $\boldsymbol{V}_3^T \boldsymbol{V}_3 = \boldsymbol{I}_3$, donde $\boldsymbol{I}_3$ es la matriz identidad de orden 3. Entonces

$$\boldsymbol{V}_3^T \boldsymbol{v}_1 = \begin{bmatrix} \boldsymbol{v}_1^T \boldsymbol{v}_1 \\ \boldsymbol{v}_2^T \boldsymbol{v}_1 \\ \boldsymbol{v}_3^T \boldsymbol{v}_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \boldsymbol{e}_1^3 \ ; \tag{3.13}$$

donde $\boldsymbol{e}_1^3 = (1, 0, 0)^T \in \mathrm{R}^{3\times 1}$ cuando $m = 3$. Por inducción sobre m podemos generalizar la Ec. (3.13) resultando

$$\begin{aligned} \boldsymbol{V}_m^T \boldsymbol{v}_1 &= \boldsymbol{e}_1^m \ ; \\ \boldsymbol{V}_{m+1}^T \boldsymbol{v}_1 &= \boldsymbol{e}_1^{m+1} \ . \end{aligned} \tag{3.14}$$

Ahora hacemos, por una parte

$$\boldsymbol{X} = \boldsymbol{A}\boldsymbol{V}_3 = \boldsymbol{A} \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 & \boldsymbol{v}_3 \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}\boldsymbol{v}_1 & \boldsymbol{A}\boldsymbol{v}_2 & \boldsymbol{A}\boldsymbol{v}_3 \end{bmatrix} \ ; \tag{3.15}$$

y por otra

$$\begin{aligned} \boldsymbol{Y} = \boldsymbol{V}_4 \tilde{\boldsymbol{H}}_3 &= \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 & \boldsymbol{v}_3 & \boldsymbol{v}_4 \end{bmatrix} \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ \beta_1 & h_{22} & h_{23} \\ 0 & \beta_2 & h_{33} \\ 0 & 0 & \beta_3 \end{bmatrix} \\ &= \left[\ (\boldsymbol{v}_1 h_{11} + \boldsymbol{v}_2 \beta_1) \quad (\boldsymbol{v}_1 h_{12} + \boldsymbol{v}_2 h_{22} + \boldsymbol{v}_3 \beta_2) \quad (\boldsymbol{v}_1 h_{13} + \boldsymbol{v}_2 h_{23} + \boldsymbol{v}_3 h_{33} + \boldsymbol{v}_4 \beta_3) \ \right] \ . \end{aligned} \tag{3.16}$$

Si planteamos $\boldsymbol{X} = \boldsymbol{Y}$ resultan las relaciones

$$\begin{aligned} \boldsymbol{A}\boldsymbol{v}_1 &= \boldsymbol{v}_1 h_{11} + \boldsymbol{v}_2 \beta_1 \ ; \\ \boldsymbol{A}\boldsymbol{v}_2 &= \boldsymbol{v}_1 h_{12} + \boldsymbol{v}_2 h_{22} + \boldsymbol{v}_3 \beta_2 \ ; \\ \boldsymbol{A}\boldsymbol{v}_3 &= \boldsymbol{v}_1 h_{13} + \boldsymbol{v}_2 h_{23} + \boldsymbol{v}_3 h_{33} + \boldsymbol{v}_4 \beta_3 \ ; \end{aligned} \tag{3.17}$$

es decir,

$$\begin{aligned} \boldsymbol{v}_2 &= (\boldsymbol{A}\boldsymbol{v}_1 - \boldsymbol{v}_1 h_{11})/\beta_1 \ ; \\ \boldsymbol{v}_3 &= (\boldsymbol{A}\boldsymbol{v}_2 - \boldsymbol{v}_1 h_{12} - \boldsymbol{v}_2 h_{22})/\beta_2 \ ; \\ \boldsymbol{v}_4 &= (\boldsymbol{A}\boldsymbol{v}_3 - \boldsymbol{v}_1 h_{13} - \boldsymbol{v}_2 h_{23} - \boldsymbol{v}_3 h_{33})/\beta_3 \ ; \end{aligned} \tag{3.18}$$

que no son otra cosa que las relaciones dadas por el algoritmo de Arnoldi. Por último, también es inmediato verificar que

$$\beta_3 \boldsymbol{v}_4 (\boldsymbol{e}_3^3)^T = \beta_3 \boldsymbol{v}_4 \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{o} & \mathbf{o} & \beta_3 \boldsymbol{v}_4 \end{bmatrix} \ . \tag{3.19}$$

Teniendo en cuenta las (3.15,3.16,3.19) tenemos que

$$\begin{aligned} \boldsymbol{A}\boldsymbol{V}_3 &= \boldsymbol{V}_4\tilde{\boldsymbol{H}}_3 \ ; \\ \boldsymbol{A}\boldsymbol{V}_3 &= \boldsymbol{V}_3\boldsymbol{H}_3 + \beta_3\boldsymbol{v}_4(\boldsymbol{e}_3^3)^T \ . \end{aligned} \tag{3.20}$$

En la Ec. (3.20.a) despejamos $\tilde{\boldsymbol{H}}_3$ en forma inmediata, multiplicando miembro a miembro por $\boldsymbol{V}_4^T$ y, a continuación, usamos la propiedad $\boldsymbol{V}_4^T\boldsymbol{V}_4 = \boldsymbol{I}_4$, donde $\boldsymbol{I}_4$ es la matriz identidad de orden 4, Para despejar $\boldsymbol{H}_3$ en la Ec. (3.20.a) hacemos lo mismo pero teniendo en cuenta ahora que $\boldsymbol{V}_3^T\boldsymbol{v}_4 = \mathbf{o}$ pues, por construcción, $\boldsymbol{v}_4$ es ortogonal a $\boldsymbol{V}_3$. En definitiva, re-escribimos la Ec. (3.20) como

$$\begin{aligned} \boldsymbol{V}_4^T\boldsymbol{A}\boldsymbol{V}_3 &= \tilde{\boldsymbol{H}}_3 \ ; \\ \boldsymbol{V}_3^T\boldsymbol{A}\boldsymbol{V}_3 &= \boldsymbol{H}_3 \ ; \end{aligned} \tag{3.21}$$

que valen para $m = 3$. Entonces, por inducción sobre m, generalizamos la Ec. (3.21), resultando

$$\begin{aligned} \boldsymbol{V}_{m+1}^T\boldsymbol{A}\boldsymbol{V}_m &= \tilde{\boldsymbol{H}}_m \ ; \\ \boldsymbol{V}_m^T\boldsymbol{A}\boldsymbol{V}_m &= \boldsymbol{H}_m \ . \end{aligned} \tag{3.22}$$

Notar que $\tilde{\boldsymbol{H}}_m \in \mathrm{R}^{(m+1)\times m}$ es rectangular mientras que $\boldsymbol{H}_m \in \mathrm{R}^{m\times m}$ son las primeras m filas y columnas de $\tilde{\boldsymbol{H}}_m$. Las propiedades dadas por la Ec. 3.22 son *fundamentales* para la deducción de los algoritmos FOM y GMRES.

3.21.3. Full Orthogonalization Method (FOM)

Para resolver aproximadamente el SEL $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ buscamos la solución iterada $\boldsymbol{x}_m$ en la forma

$$\begin{aligned} \boldsymbol{x}_m &= \boldsymbol{x}_0 + \boldsymbol{z}_m \ ; \\ \boldsymbol{z}_m &= \boldsymbol{V}_m\boldsymbol{y}_m \ ; \end{aligned} \tag{3.23}$$

con $\boldsymbol{x}_0 \in \mathrm{R}^{n\times 1}$ como una solución iterada inicial, $\boldsymbol{z}_m \in \mathrm{R}^{n\times 1}$ es un vector correctivo, $\boldsymbol{y}_m \in \mathrm{R}^{m\times 1}$ es un vector auxiliar y $\boldsymbol{V}_m \in \mathrm{R}^{n\times m}$ es la base ortonormal del subespacio de Krylov $\mathrm{K}_m = \mathrm{span}\{\boldsymbol{v}_1, \boldsymbol{A}\boldsymbol{v}_1, ..., \boldsymbol{A}^{m-1}\boldsymbol{v}_1\}$ inducido por el vector inicial $\boldsymbol{v}_1 = \boldsymbol{r}_0/\beta_0$, donde $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$ es el residuo inicial, $\beta_0 = ||\boldsymbol{r}_0||_2$, y que es obtenido con el algoritmo de Arnoldi. El método de Galerkin impone que el residuo iterado $\boldsymbol{r}_m$ sea ortogonal al subespacio de Krylov. Esto se logra imponiendo la condición

$$\boldsymbol{V}_m^T\boldsymbol{r}_m = \mathbf{o} \ . \tag{3.24}$$

Entonces, usando también la Ec. 3.22,

$$\begin{aligned} \boldsymbol{V}_m^T\boldsymbol{r}_m &= \boldsymbol{V}_m^T(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m) \\ &= \boldsymbol{V}_m^T(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{A}\boldsymbol{z}_m) \\ &= \boldsymbol{V}_m^T(\boldsymbol{r}_0 - \boldsymbol{A}\boldsymbol{z}_m) \\ &= \boldsymbol{V}_m^T\boldsymbol{r}_0 - \boldsymbol{V}_m^T\boldsymbol{A}\boldsymbol{z}_m \\ &= \beta_0\boldsymbol{V}_m^T\boldsymbol{v}_1 - \boldsymbol{V}_m^T\boldsymbol{A}\boldsymbol{V}_m\boldsymbol{y}_m \\ &= \beta_0\boldsymbol{e}_1^m - \boldsymbol{H}_m\boldsymbol{y}_m \\ &= \mathbf{o} \ . \end{aligned} \tag{3.25}$$

Entonces, el vector auxiliar $\boldsymbol{y}_m$ se obtiene resolviendo el SEL *cuadrado*

$$\boldsymbol{H}_m \boldsymbol{y}_m = \beta_0 \boldsymbol{e}_1^m \; ; \tag{3.26}$$

lo cual da lugar al FOM propuesto por Saad (1981) [35].

3.21.4. Generalized Minimal RESidual (GMRES)

Después de m pasos del algoritmo de Arnoldi obtenemos la base ortonormal $\boldsymbol{V}_{m+1}$ y la matriz $\tilde{\boldsymbol{H}}_m$. Entonces, con la Ec. 3.22,

$$\begin{aligned} \boldsymbol{r}_m &= \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m \\ &= \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{A}\boldsymbol{z}_m \\ &= \boldsymbol{r}_0 - \boldsymbol{A}\boldsymbol{z}_m \\ &= \boldsymbol{r}_0 - \boldsymbol{A}\boldsymbol{V}_m\boldsymbol{y}_m \\ &= \beta_0\boldsymbol{v}_1 - \boldsymbol{V}_{m+1}\tilde{\boldsymbol{H}}_m\boldsymbol{y}_m \\ &= \boldsymbol{V}_{m+1}(\beta_0\boldsymbol{e}_1^{m+1} - \tilde{\boldsymbol{H}}_m\boldsymbol{y}_m) \; . \end{aligned} \tag{3.27}$$

La norma l_2 de este residuo es

$$\begin{aligned} ||\boldsymbol{r}_m||_2 &= ||\boldsymbol{V}_{m+1}(\beta_0\boldsymbol{e}_1^{m+1} - \tilde{\boldsymbol{H}}_m\boldsymbol{y}_m)||_2 \\ &= ||\beta_0\boldsymbol{e}_1^{m+1} - \tilde{\boldsymbol{H}}_m\boldsymbol{y}_m||_2 \; ; \end{aligned} \tag{3.28}$$

donde se omite la base $\boldsymbol{V}_{m+1}$ porque al ser ortonormal también se cumple que $\det(\boldsymbol{V}_{\mathrm{m+1}}) = 1$, por lo que no incide en la búsqueda del mínimo de la norma l_2 del vector residuo $\boldsymbol{r}_m$. Entonces, ahora $\boldsymbol{y}_m$ es el mínimo del sistema *rectangular*

$$\tilde{\boldsymbol{H}}_m \boldsymbol{y}_m = \beta_0 \boldsymbol{e}_1^{m+1} \; ; \tag{3.29}$$

lo cual da lugar al GMRES que es extensivamente empleado y que fue propuesto por Saad-Schultz (1986) [36].

Capítulo 4

Ejercicios

Sugerencias: cuando corresponda, implementar o testear los programas demos. Eventualmente, incluir pruebas de escritorio para justificar los resultados obtenidos por los mismos. Prestar atención a:

- la sintaxis matricial del lenguaje;
- las operaciones de dispersión de índices;
- las operaciones con matrices ralas en los formatos: *completo, CSC y CSR.*

1. [**instalaciones bajo Linux**]. Instalar, al menos en una única PC, los siguientes paquetes de distribución libre bajo Linux:

   a) los compiladores: `ifort` (versión 9) y `g95`, o bien el `gfortran` en lugar del segundo. Notar que el `g95` es *gnu-based* mientras que el `gfortran` es *gnu-native*;
   b) la librería de paso de mensajes MPICH2 versión 1.0.4 (o en su defecto la 1.0.3) pero *incluyendo* la parte de Fortran 90;
   c) el traductor `ADAPTOR` versión 10.1 .

2. [**intrínsecas** *lbound, ubound, size, shape*]. Usando las intrínsecas matriciales `lbound`, `ubound`, `size` y `shape`, determinar los extremos inferior y superior, el tamaño y la forma de cada uno de los siguientes arreglos:

```
real   , dimension (1:10)     :: a
real   , dimension (2,0:2)    :: b
integer, dimension (-1:1,3,2) :: c
real   , dimension (0:1,3)    :: d
```

3. [**intrínseca** *where ... else where*]:

   a) Para un arreglo de 2 índices `a (i,j)` de enteros, escribir una instrucción matricial `where` en donde se anulan todas sus entradas `a (i,j)` impares;
   b) Para un arreglo de 2 índices `a (i,j)` de reales, escribir una instrucción matricial `where else where` en donde, si la entrada `a (i,j)` es positiva, entonces calcular su raíz cuadrada y si no, anularla.

4. [**intrínsecas** *cshift, eoshift*]. Para los arreglos

$$\texttt{a} = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \quad ; \quad \texttt{b} = \begin{bmatrix} 9 & 8 & 2 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix} ; \tag{4.1}$$

determinar el resultado de las siguientes instrucciones de desplazamiento circular y lineal: (i) `cshift (a,shift=-1)`; (ii) `cshift (a,shift=+3)`; (iii) `eoshift (b,shift=+1,dim=1)`; (iv) `eoshift (b,shift=-1,dim=2)`.

5. [**intrínsecas** *maxloc, maxval, minloc, minval, count*]. Para el arreglo

$$\texttt{a} = \begin{bmatrix} 3 & -4 & 8 & 6 \\ 1 & 3 & 5 & 4 \\ 0 & 2 & -5 & -1 \end{bmatrix} ; \tag{4.2}$$

determinar el resultado de las siguientes instrucciones de reducción: (i) `maxloc (a)`; (ii) `maxval (a)`; (iii) `minloc (a,a<5)`; (iv) `minval (a,a<5)`; (v) `count (a,a>0)`; (vi) `count (a,a<0)`.

6. [**intrínseca** *reshape*]. Reacomodar el vector

$$\texttt{v} = \begin{bmatrix} 3 & 1 & 0 & -4 & 3 & 2 & 8 & 5 & -5 & 6 & 4 & -1 \end{bmatrix} ; \tag{4.3}$$

como el arreglo

$$\texttt{a} = \begin{bmatrix} 3 & -4 & 8 & 6 \\ 1 & 3 & 5 & 4 \\ 0 & 2 & -5 & -1 \end{bmatrix} ; \tag{4.4}$$

mediante la instrucción `reshape (v, forma)`, donde `forma` es un vector con el número de filas y columnas del arreglo `a`.

7. [**intrínseca** *merge*]. Determinar el efecto de la instrucción `merge (a, b, c)`, cuando

$$\texttt{a} = \begin{bmatrix} 2 & 4 & 6 \\ 8 & 6 & 4 \end{bmatrix} \quad ; \quad \texttt{b} = \begin{bmatrix} 1 & 3 & 5 \\ 7 & 9 & 3 \end{bmatrix} \quad ; \quad \texttt{c} = \begin{bmatrix} T & T & F \\ F & T & F \end{bmatrix} . \tag{4.5}$$

8. [**intrínseca** *pack*]. Considere las matrices

$$\texttt{a} = \begin{bmatrix} 0 & 5 & 0 \\ 1 & 0 & 9 \end{bmatrix} \quad ; \quad \texttt{b} = \begin{bmatrix} 7 & -2 & 0 \\ -3 & -4 & 5 \end{bmatrix} ; \tag{4.6}$$

y los vectores

$$\texttt{v} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \quad ; \quad \texttt{z} = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} . \tag{4.7}$$

Determinar el resultado de las siguientes instrucciones: (i) `pack (a, a.ne.0)`; (ii) `pack (b, b>0)`; (iii) `pack (a, a.ne.0, vector=v)`; (iv) `pack (b, a>0, vector=z)`.

9. [**intrínseca** *unpack*]. Considere los arreglos

$$\mathtt{a} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad ; \quad \mathtt{b} = \begin{bmatrix} F & T & F \\ T & F & F \\ F & F & T \end{bmatrix} \quad ; \quad \mathtt{c} = \begin{bmatrix} 4 & 5 & 6 & 7 & 8 \end{bmatrix} . \tag{4.8}$$

   Determinar el resultado de las siguientes instrucciones: (i) `unpack (c, mask = b, field = -3)`; (ii) `unpack (c, mask = b, field = a)`.

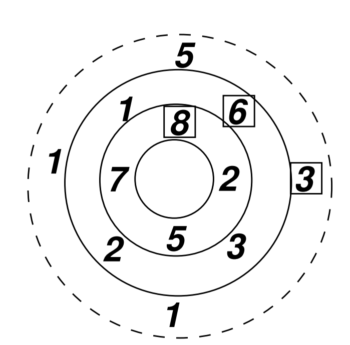
10. [**instrucciones** *do, forall*]. Considere el arreglo `a` = [11 22 33 44 55], de tamaño `n = size (a)`. Determine el resultado de las siguientes instrucciones:

```
do      i = 2 , n  ; a (i) = a (i-1) ; end do
forall (i = 2 : n)   a (i) = a (i-1)
```

11. [**intrínseca** *spread*]. Escriba una función matricial F95 que devuelva el producto *exterior* de los vectores `a` y `b`, de tamaño `n = size (a) = size (b)`. Ayuda: considere la intrínseca `spread`. El resultado es una matriz `c` tal que `c (i,j) = a (i) * b (j)`, para todo `i,j`.

12. [**promedio circular** (secuencial)]. Dado el arreglo $\boldsymbol{a} = (a_1, a_2, ..., a_n)$ escriba un demo en F95 que devuelva otro arreglo $\boldsymbol{b} = (b_1, b_2, ..., b_n)$ con el promedio de los elementos adyacentes de **a**, i.e. $b_i = (a_{i-1} + a_{i+1})/2$, para todo i. En el b_1 usar el segundo y el último elemento de $\boldsymbol{a}$, mientras que en el b_n usar el primer y penúltimo elemento de $\boldsymbol{a}$ [Ayuda: considere la intrínseca `cshift`].

13. [**diferencia circular** (secuencial)]. Coloquemos n números enteros positivos alrededor de una circunferencia. Construyamos ahora sucesivas circunferencias concéntricas hacia el exterior, de igual cantidad de elementos, los cuales son obtenidos restando en valor absoluto pares de consecutivos en circunferencia actual más externa. Puede demostrarse que si $n = 2^k$ en alguna iteración p aparecerán n números iguales y, en ese momento, se finalizan las iteraciones. Por ejemplo, supongamos $k = 2$, $(n = 4)$ y que la circunferencia "inicial" sea $C_0 = (8, 2, 5, 7)$, entonces iteramos y obtendremos sucesivamente, $C_1 = (6, 3, 2, 1)$, $C_2 = (3, 1, 1, 5)$, $C_3 = (2, 0, 4, 2)$, $C_4 = (2, 4, 2, 0)$ y $C_5 = (2, 2, 2, 2)$, por lo que el número de circunferencias iteradas es $p = 5$. Entonces, dado un arreglo $a = [a_1, a_2, ..., a_n]$ de n números enteros que representan los valores alrededor de la circunferencia inicial, escribir un demo en F95 que ejecute esta tarea, dando además el número de circunferencias iteradas p.



   Restricciones: el algoritmo debe ser *in place* y debe incluir un control *simple* (e.g. un `if` y una cuenta) que verifique si n es o no una potencia de 2. *Ayuda:* pensar al arreglo en un *"sentido circular"* pero tener cuidado al generar la diferencia correspondiente a los extremos; para un control *simple* de si n es o no una potencia de 2, tenga presente la representación binaria de n y de $n - 1$ y alguna operación de bits, bit a bit.

14. [**realoca memoria RAM** (secuencial)]. Escriba un demo secuencial en F95 para alocar y realocar memoria RAM usando variables y funciones que tengan el atributo de punteros.

15. [**producto matriz vector** (secuencial)]. Usando compiladores con diversos niveles de optimización, escriba un test secuencial en F95 para medir los tiempos en una multiplicación matricial usando:

    a) lazos `do` con distintos anidamientos;
    b) la intrínseca `matmult`
    c) la intrínseca `dot_product`
    d) la subrutina `sgemm` de la *Basic Linear Algebra Subprograms* (BLAS).

16. [**dispersión de índices** (secuencial)]. Determine el resultado de las dispersiones de índices dadas por `b=a(i)`, `c=a(j)` y `d(i)= a`, cuando

$$\begin{aligned} a &= \begin{bmatrix} 7 & 3 & 5 & 4 & 1 & 8 \end{bmatrix} ; \\ i &= \begin{bmatrix} 4 & 6 & 1 & 3 & 5 & 2 \end{bmatrix} ; \\ j &= \begin{bmatrix} 2 & 1 & 1 & 3 & 2 & 3 \end{bmatrix} . \end{aligned} \tag{4.9}$$

17. [**formatos ralos completo, CSC y CSR**]. Suponga que se pide evaluar el producto matricial $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ dado por

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 8 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 6 & 9 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} . \tag{4.10}$$

    Para evaluar el producto $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ considere las alternativas dadas en la Sec. 3.16, cuando la matriz $\boldsymbol{A}$ se guarda como:

    a) rala en formato ralo completo (“inocente”);
    b) rala en formato CSC (Compressed Sparse Column);
    c) rala en formato CSR (Compressed Sparse Row);

    Entonces:

    a) Muestre los arreglos de almacenamiento involucrados en cada formato;
    b) Haga una prueba de escritorio, para cada formato, mostrando los movimientos de índices (dispersiones y reducciones). Verifique sus resultados realizando el producto usual con la matriz densa;

18. [**producto matriz-vector** con matriz banda almacenada por diagonales (secuencial)]. Escriba un demo secuencial con módulos en F95 que, dada una matriz banda en formato comprimido A_c de $n \times m$, y el vector x de $n \times 1$, devuelva otro vector b con el producto matriz-vector, usando intrínsecas tales como `eoshift`, `spread`, `sum` o `size`, donde $m = m_1 + 1 + m_2$ es el ancho de la banda, primero con las m_1 sub-diagonales, luego la diagonal principal y, finalmente, las m_2 supra-diagonales, como columnas consecutivas en la matriz en formato comprimido A_c.

19. [**factorización LU** (secuencial)]. Dados el Sistema de Ecuaciones Lineales (SEL) de la forma $\boldsymbol{Ax} = \boldsymbol{b}$, o bien $\boldsymbol{AX} = \boldsymbol{B}$, donde se asume que la matriz del sistema $\boldsymbol{A}$ es regular y densa, escribir un módulo secuencial `m_gauss0.f90` en F95 que los resuelva numéricamente en forma directa, realizando:

    a) factorización LU con alguna estrategia de pivotaje;
    b) resolución hacia adelante;
    c) resolución hacia atrás.

    Codificar cada una de estas tareas como procedimientos genéricos con argumentos de tipo real o complejo, usando precisión doble y extendida en cada caso. Finalmente, escriba demos secuenciales para testear el módulo para cada uno de los casos previstos. Observaciones:

    a) con el `g95` sólo es posible la precisión extendida `real *10` (no estándar) mientras que con el `ifort` es posible la precisión **cuádruple**.
    b) en el caso con variable compleja sería mejor evitar el artilugio de convertir el SEL dado en otro equivalente en variable real, e.g. ver discusión en el *Numerical Recipes*, por la memoria RAM y número de operaciones.

20. [**solución iterativa de un SEL por el FOM** (secuencial)]. Dado el SEL $\boldsymbol{Ax} = \boldsymbol{b}$, en donde se asume que la matriz del sistema $\boldsymbol{A}$ es regular y densa, escribir un módulo `m_krylov0.f90` en lenguaje F95 secuencial que lo resuelva numéricamente en forma iterativa, incluyendo:

    a) el algoritmo de Arnoldi para hallar una base ortonormal $\boldsymbol{V}, \boldsymbol{H}$ del subespacio de Krylov a partir de un residuo inicial $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{Ax}_0$, donde $\boldsymbol{x}_0$ es una solución iterada inicial arbitraria;
    b) el método de ortogonalización completa (*Full Orthogonalization Method*, FOM) propuesto por Saad [35].

    Codificar cada una de estas tareas como procedimientos genéricos con argumentos de tipo real, usando precisión doble y extendida, y escriba demos para testear el módulo en cada uno de los casos previstos.

21. [**cómputo distribuido de pi**]. Sabemos que si $y = \text{atan}(x)$ entonces su derivada es $y' = 1/(1 + x^2)$, con $\text{atan}(1) = \pi/4$ y $\text{atan}(0)$, por lo que un método para obtener π es calcular la integral definida

    $$\pi = 4 \int_0^1 \frac{dx}{1 + x^2} \; . \tag{4.11}$$

    Entonces, escriba un demo de cómputo distribuido con HPF-F95 que aproxime el valor de π usando la regla del punto medio compuesta

    $$\int_a^b f(x) dx \approx \sum_{i=1}^{n} h f(a + h(i - 1/2)) \quad ; \quad h = (b - a)/n \; ; \tag{4.12}$$

    en donde, en este caso, $f(x) = 1/(1 + x^2)$, $a = 0$ y $b = 1$.

22. [**conteo distribuido de números primos en un intervalo**]. Escriba un demo de cómputo distribuido con HPF-F95 para contar la cantidad de números de primos que hay en el rango 2 a n usando implementaciones:

    a) una básica pero con alguna distribución;
    b) mejorada con cómputo replicado hasta `sqrt (n)`;
    c) que incluya un procedimiento `HPF local`.

23. [**producto distribuido matriz densa por vector**]. Escriba un demo de cómputo distribuido con HPF-F95 que incluya módulos para calcular los productos: (i) matriz por vector $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$; y (ii) matriz traspuesta por vector $\boldsymbol{y} = \boldsymbol{A}'\boldsymbol{x}$, cuando la matriz $\boldsymbol{A}$ es *densa*.

24. [**productos distribuidos matriz rala por vector**]. Escriba un demo de cómputo distribuido con HPF-F95 que incluya módulos para calcular los productos: (i) matriz por vector $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$; y (ii) matriz-traspuesta por vector $\boldsymbol{y} = \boldsymbol{A}'\boldsymbol{x}$, cuando la matriz $\boldsymbol{A}$ es *rala*, en algún formato de su elección.

25. [**factorización LU** (distribuido)]. Dados el Sistema de Ecuaciones Lineales (SEL) de la forma $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, o bien $\boldsymbol{A}\boldsymbol{X} = \boldsymbol{B}$, en donde se asume que la matriz del sistema $\boldsymbol{A}$ es regular y densa, escribir un módulo distribuido `m_gauss1.f90` en HPF-F95 que los resuelva numéricamente en forma directa, realizando:

    a) factorización LU con alguna estrategia de pivotaje;
    b) resolución hacia adelante;
    c) resolución hacia atrás.

    Codificar cada una de estas tareas como procedimientos genéricos con argumentos de tipo real o complejo usando precisión doble. En el caso con variable compleja sería mejor evitar el artilugio de convertir el SEL dado en otro equivalente en variable real, e.g. ver discusión en el *Numerical Recipes*, por la memoria RAM y número de operaciones, además de problemas de alinemientos de los arreglos distribuidos. Finalmente, escriba demos para testear el módulo en cada uno de los casos previstos, trazando curvas de rendimiento de *speedup* y eficiencia.

26. [**solución iterativa de un SEL por el FOM** (distribuido)]. Dado el SEL $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, en donde se asume que la matriz del sistema $\boldsymbol{A}$ es regular y densa, escribir un módulo `m_krylov1.f90` en HPF-F95 que lo resuelva numéricamente en forma iterativa, incluyendo:

    a) el algoritmo de Arnoldi para hallar una base ortonormal $\boldsymbol{V}, \boldsymbol{H}$ del subespacio de Krylov a partir de un residuo inicial $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$, donde $\boldsymbol{x}_0$ es una solución iterada inicial arbitraria;
    b) el método de ortogonalización completa (*Full Orthogonalization Method*, FOM) propuesto por Saad [35].

    Codificar cada una de estas tareas como procedimientos genéricos con argumentos de tipo real usando precisión doble. Finalmente, escriba demos para testear el módulo en cada uno de los casos previstos, trazando curvas de rendimiento de *speedup* y eficiencia.

Apéndice A

GNU Free Documentation License

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s

overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all

these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.
You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.
You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another;

but you may replace the old one, on explicit permission from the previous publisher that added the old one.
The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.
The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.
In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.
You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document. If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.
Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> *Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".*

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts. If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliografía

[1] J. Arndt. *Algorithms for programmers.* http://www.jjj.de/fxt/, 2003.

[2] W.E. Arnoldi. The principle of minimized iteration in the solution of the matrix eigenvalue problems. *Quart. Appl. Math.*, 9:17–29, 1951.

[3] V. Blanco, J.C. Cabaleiro, P. González, D.B. Heras, T.F. Pena, J. Pombo, and F.F. Rivera. Diseño y análisis de métodos iterativos en HPF. In *X Jornadas de Paralelismo*, La Manga del Mar Menor, Murcia, september 1999.

[4] T. Brandes. ADAPTOR: Parallel fortran compilation system. http://www.scai.fraunhofer.de/EP-CACHE/adaptor/www/adaptor_home.html.

[5] T. Brandes. ADAPTOR: Parallel fortran compilation system: Installation and users guides, OpenMP and HPF programmers guides and HPF language reference manual. http://www.scai.fraunhofer.de/EP-CACHE/adaptor/www/adaptor_home.html.

[6] E. Chu and A. George. *Inside the FFT black box.* CRC Press, 2000.

[7] P. Corde and H. Delouis. Cours Fortran 95, version 9.1. Technical report, Institut du Développement et des Ressources en Informatique Scientifique, June 2005. http://webserv2.idris.fr, Cours de l'IDRIS, Prochains cours.

[8] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1997.

[9] J.L. De la Fuente O'Connor. *Técnicas de Cálculo para Sistemas de Ecuaciones, Programación Lineal y Programación Entera.* Reverté, 1998.

[10] V.K. Decyk. How to Express C++ Concepts in Fortran 90. *Scientific Programming*, 6(4):363–390, 1997. IOS Press.

[11] A.K. Ewing, R.J. Hare, H. Richardson, and A.D. Simpson. Writing Data Parallel Programmes with High Perfomance Fortran. Technical report, The University of Edinburgh, 1999.

[12] NAGWare f95 compiler. http://www.nag.co.uk.

[13] Co-array Fortran. http://www.co-array.org/.

[14] G95: a GNU-based Fortran 95 compiler. http://g95.sourceforge.net/.

[15] Intel C++ and Fortran compilers. http://www.intel.com/cd/software/products/asmo-na/eng/compilers.

[16] Open Machine Parallel, version 2.5: combined c/c++ and fortran specification, may 2005). http://www.openmp.org/.

[17] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[18] A. George and J.W. Liu. *Computer solution of large sparse positive systems*. Prentice-Hall, 1981.

[19] P. Graham. Openmp course. Technical report, The University of Edinburgh, 1999.

[20] A. Grama, A. Grupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2 edition, 2003.

[21] M. Hermans. Parallel programming in fortran 95 using OpenMP. Univ. Polit. de Madrid, 2002.

[22] FHPF (Japan): free HPF translator on Linux and Solaris. http://www.hpfpc.org/fhpf-E.html.

[23] G.E. Karniadakis and R.M. Kirby II. *Parallel Scientific Computing in C++ and MPI*. Cambridge University Press, 2003.

[24] C.H. Koebel, D.B. Loveman, G.L. Steele, and M.E. Zosel. *The High Perfomance Fortran Handbook*. The MIT Press, 1997.

[25] Linux Doc. Project, http://sunsite.unc.edu/mdw/linux.html.

[26] A.C. Marshall. HPF Programming Course Notes. Technical report, Liverpool University, 1997. http://www.liv.ac.uk/HPC/HTMLFrontPageHPF.html.

[27] A.C. Marshall, J.S. Morgan, and J.L. Schonfelder. Fortran 90 Course Notes. Technical report, Liverpool University, 1997. http://www.liv.ac.uk/HPC/HTMLFrontPageF90.html.

[28] A.C. Marshall and J.L. Schonfelder. Programming in Fortran 90/95. Technical report, Liverpool University, 2000.

[29] L.P. Meissner. Fortran 90 and 95. Array and Pointer Techniques. Objects, Data Structures and Algorithms. Technical report, Computer Science Department. University of San Francisco, 1998.

[30] C.D. Norton. *Object Oriented Programming Paradigms in Scientific Computing*. PhD thesis, Rensselaer Polytechnic Institute, 1996.

[31] B. Parhami. *Introduction to Parallel Processing*. Kluwer Academic Publishers, 2002. eBook.

[32] S. Pissanetzky. *Sparse matrix technology*. Academic Press, 1984.

[33] *Portland Cluster Kit: C++/F95/HPF compilers*. http://www.pgroup.com/.

[34] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes*. Cambridge University Press, 2nd edition, 1992.

BIBLIOGRAFÍA

[35] Y Saad. Krylov subspace methods for solving large unsymmetric linear systems. *SIAM Math. Comput.*, 37:105–126, 1981.

[36] Y. Saad and M. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.

[37] B.K. Szymanski. Mistakes in fortran 90 programs that might surprise you. http://www.cs.rpi.edu/ szymansk/OOF90/bugs.html.